MASTER IN COMPUTER SCIENCE

Authentication and non-repudiation of application level real-time flows

Robinet, William

Award date:
2002

FUNDP
Institut d'Informatique

Rue Grandgagnage, 21
B - 5000 NAMUR (Belgique)

Authentication and non-repudiation of application level real-time flows.

William Robinet

Promoteur : Olivier Bonaventure
(FUNDP - Institut d'Informatique, Namur)

Mémoire présenté en vue de l'obtention
du grade de maître en informatique

Septembre 2002

Abstract

In this paper we firstly give an overview of the existing techniques for the authentication and non-repudiation of real-time flows at the application level. Next, we propose an original Java implementation of these different techniques under the form of a prototype allowing to exchange, record and authenticate real-time video data. This implementation is based on the Java Media Framework extension and make use of its plugin architecture. Finally, we propose a performance evaluation of the different authentication schemes we implemented.

Keywords: real-time flow, authentication, non-repudiation, application level, Java Media Framework

Dans ce mémoire, nous donnons, dans un premier temps, un aperçu des techniques existantes permettant l'authentification et la non-répudiation de flux temps réel au niveau applicatif. Nous proposons ensuite une implémentation Java originale des différentes méthodes recensées sous la forme d'un prototype fonctionnel permettant l'échange, l'enregistrement et l'auhtentification de données video en temps réel. Cette implementation est basée sur l'extension Java Media Framework et fait usage de son architecture de plugins. Nous proposons finalement une évaluation des performances des differentes méthodes d'authentification que nous avons implémentées.

Mots clés: flux temps réel, authentification, non-répudiation, niveau applicatif, Java Media Framework

Acknowledgments

First I would like to thank my promoter Olivier Bonaventure for his patience and his precious help during this year of work.

Thanks to the people at Lancaster University who helped me during my training period, and especially to Laurent Mathy with whom I learned so much.

Thanks to Caroline, my family and my friends for their help and support during the hard moments.

And finally, thanks to all the people who will read this paper and be present during my presentation.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The growing utilization of the Internet implies that more and more every day life activities are transposed to their online equivalents. This is particularly the case for the world of commerce for which these changes constitute the advent of new market places. In this context, the traditional contracts have to be adapted to suit new needs.

The constraints undergone by electronic contracts are not the same as the ones which are undergone by their traditional equivalents. Indeed, these latter are based on a handwritten component which is recognized as an evidence by the laws. All the difficulty in developing electronic contracts resides in the problem of finding an electronic equivalent to this missing component.

Solutions based on public-key cryptography have already been developed. They provide signature techniques allowing to meet the authenticability requirements of electronic contents. In order to be signed, the considered electronic content must be entirely known at the time the signature is appended to it.

With the development of interactive multimedia applications, it is now possible for future contractors to meet each other on the Internet. The audio and video communications which are performed can be recorded. At anytime, it must be possible to authenticate their content. It means that the identity of the source and the integrity of the exchanged data must be verifiable at any time, either during the communication itself or on its recording.

Applications allowing to perform such audio/video communications are based on real-time flows. This kind of flows makes use of the RTP protocol which is itself based on the non-reliable connectionless UDP protocol. The authentication mechanisms have to be built on top of it. This has to be done by taking into account the negative characteristics of the Internet.

Indeed, the Internet is known to be a lossy and insecure network. Its lossy character is a direct consequence of the techniques involved in the packet

routing process. Most of the Internet traffic is ruled by the best effort quality of service. It means that the network does not provide any guarantee on its ability to transport a packet from a given point to another.

It must also be considered as an insecure network. It does not provide confidentiality or integrity assurances of any kind. Also, there is no guarantee on the identity of the source of the received packets. These security issues must be considered with attention in the process of building a critical application which makes use of real-time flows. In absence of adequate mechanisms, it is impossible for this kind of applications to have any assurance about the received packet. Furthermore, the lossy character of the Internet amplify the difficulty to find efficient solutions to these problems.

The objective of this study is to implement and evaluate different real-time flow authentication and non-repudiation mechanisms. We begin with a description of the problem from a technical point of view. We then present the existing solutions. We expose their characteristics and the way they work. We end this presentation by a comparison of all of them.

After this, we present our implementation of some of these authentication schemes under the form of a prototype allowing to perform the exchange and the authentication of real-time video data. The authentication is allowed during a communication or can be performed on recordings of a past communication.

Finally, we present an evaluation of different characteristics of the authentication schemes we implemented.

Part I

Foundations

Chapter 2

Problem presentation

2.1 Introduction

In this chapter we describe the environment in which this study has been led. We expose the basic network and cryptographic knowledges necessary for the comprehension of the following chapters. We then highlight the constraints relevant to the flow authentication problem.

2.2 Network environment

As stated in the opening chapter, the objective of this study is to implement and evaluate different real-time flow authentication and non-repudiation mechanisms. In this section, we define what real-time flows are, their characteristics and the constraints they have to deal with in order to be efficient. Next we give an overview of the RTP protocol.

2.2.1 Real-time flows

Real-time flows are used to transmit real-time data over a network. By real-time data, we mean data which are not entirely known before the beginning of the transmission. This includes, for instance, the data coming out of a digital video capture device used by a video conferencing application. Opposed to real-time generated data, we have already known data, which can be, for example, the content of a file.

The transmission constraints over real-time and non real-time flows are different. For non real-time transmissions, the main goal is to have a reliable data exchange, even over a lossy network. No losses are tolerated at the application level. In order to have the reliability property when losses occur,

retransmissions are needed and thus, the communication can be slow. For example, this is what is done in file transfer applications. This kind of applications can rely on TCP to transport the data. Indeed, TCP is a reliable transport protocol that provides mechanisms to control the packet flow when losses occur in case of congestion over the considered network.

For real-time transmissions, the priority is for the data to arrive on time at the receiver's side. Indeed, real-time data flows are used by applications, such as for instance those performing interactive video conferencing, which are delay-sensitive by definition. This kind of application has to present the received information periodically to the user. If some information is late, the application might not be able to present it in time. Thus, in order to keep a continuous presentation, they are replaced by dummy information. When received, the late information is then considered to be expired and thus is simply ignored. If some information is lost during the transmission, the application must have a mean to detect it and act in consequence by replacing it, again, with dummy information.

Using TCP for this kind of flows would not be appropriate since TCP will slow down a transmission each time it detects congestion. This might result into information dropping by the application, if the delay introduced by TCP is too large. The UDP transport protocol is more adequate but is missing some properties such as datagrams ordering and non-duplication guarantee. We'll see how these problems are solved with the RTP protocol, into the next section.

Another point we have to consider, is multi-receivers transmission. For, instance, a video conferencing application might be deployed between one sender and a set of receivers. Implementing this with TCP would lead in a large waste of bandwidth since, we would need as many TCP connections as there are receivers. UDP is again a better solution, since it supports multicast transmissions.

The packetization process is the responsibility of the sending application. This must be done in such a way that the receiving application can rebuild meaningful information from the received datagrams even if some of them are lost during the transmission. The information unit contained inside a single datagram is called an *application data unit*[1].

Because of the different nature of real-time and non real-time flows, applying an authentication mechanism in the same way on both of them can be inappropriate. In the case of non real-time flows, the data is usually known before the beginning of the transmission. Thus, the sender can apply a signature mechanism on the whole data, packetize the result and send it to

[1] ADU

the receiver. The receiver has to wait for everything to be arrived before he can begin the authenticity verification process. This is different for real-time flows. As they are virtually infinite, the receiver must be able to authenticate any portion of the flow at any time during the reception. And this, even if some datagrams are lost during the transmission.

This is the main difficulty we have to deal with in order to build an efficient method for real-time flow authentication. Over the Internet, and with the lack of adequate quality of service mechanisms on the routers, the transmissions are performed with best effort quality of delivery. A consequence of this is the possibility of packet losses. This implies that during the transmission of a real-time flow having for destination a multicast group, the different receivers may receive different datagrams subsequences[2].

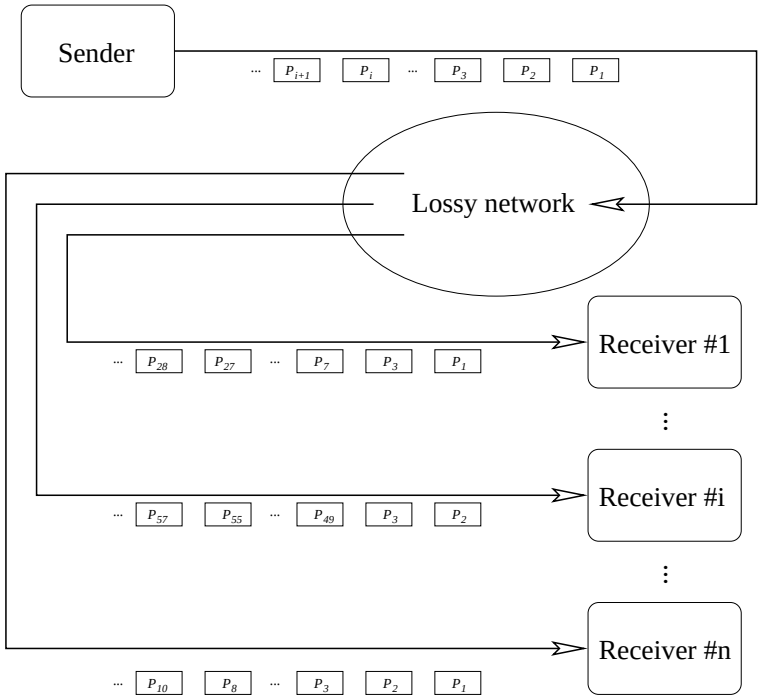

Figure 2.1: Different receivers receiving different datagrams subsequences

The other main problem with data flows occurs when the transmission is done over an insecure network. By insecure we mean a network on which the traveling datagrams can be altered by an attacker. The process of authentication should provide the guarantee of integrity for the received data. It

[2]See figure 2.1

means that if a single bit is changed inside a datagram or if a forged datagram arrives, it must be detected by the application of the receiver.

2.2.2 RTP basics

This section is based on [17] and [3].

RTP (*Real-time Transport Protocol*) along with RTCP (*Real-Time Transport Control Protocol*) is a protocol based on mechanisms allowing to deal with unicast and multicast real-time data transmission. It constitutes a framework for the development of application using real-time data flows. It is specified into [10].

RTP is typically used on top of connectionless transport protocols like UDP. It follows two of the principles given into [4]:

- Application level framing;
- Integrated layer processing.

Application level framing concept has already been evoked above. It says that the ADUs decomposition must be done by the application in such a way that each ADU taken separately must have a sense for the application. In this way, each ADU can be processed independently from the others and thus, losses can occur without interfering with the treatment of the normally received ADUs. When losses occur, there must exist a mean for the application to detect them. RTP provides this by adding sequence numbers in each datagram. This also gives a mean to reorder datagrams if it is necessary and allows duplicates detection.

The principle of *integrated layer processing* consists in the integration of the protocols on top of the network layer. This allows to perform more efficient processing of the datagrams. Indeed, instead of having to process the datagrams into different stages, only one is required. This allows to use the considered protocol into, for instance, embedded devices in which resources are limited.

Basically, RTP doesn't guarantee reliability, in time or in-order delivery and doesn't perform error treatments. It only provides mechanisms to detect these problems. It adds three main characteristics to the underlying transport protocol:

- Datagrams sequence numbering;
- Datagrams timestamping;
- Datagrams payload type identification.

As we've already said, sequence numbers are used to allow datagrams reordering and loss discovery on the receiver's side.

The addition of timestamps to the datagrams is important for the transmission of time-based media such as those we are interested in. It gives a relative time indication in relation to the flow. It is for a given datagram, the sampling time of the first byte of the ADU it contains. This time indication mapped to real wall clock time helps for the synchronization of the media presentation.

The transmission delay for a datagram is the elapsed time measured between the moment when this datagram is passed from the sender's application to the underlying transport layer and the moment when this datagram is passed from the transport layer to the receiver's application. The datagrams transmission delay is never constant over a given network, this is due to network load variations in time. The problem with this, is that the receiver's application needs to be continuously fed with ADUs. This won't be possible if the datagram transmission delay is not constant. A solution to this problem is the use of a playback buffer of a given time duration. The incoming datagrams are then simply put inside this buffer which acts as a FIFO queue. The application is then periodically fed with the present datagrams. The missing ones (i.e. lost or late) are replaced by dummy ones when they are needed by the application. The datagrams arriving on the receiver's side with a transmission delay greater than the buffer time duration, are simply ignored.

RTP can be extended with additional layers on top of it to suit the needs of a particular application. The media type of the data contained into the ADUs must then be quickly identified on the receiver's side in order to choose which extension should treat the incoming datagrams. This problem is solved by the addition of a payload type identification to the transmitted datagrams. Some already defined RTP extensions and payload identification types can be found in [9].

RTP transmissions are organized in sessions. A RTP session is identified by a multicast address and two UDP ports; one for RTP; and one for RTCP. RTCP is a protocol intended to give information about RTP sessions. It reports information about the transmission quality and about the participants into a given session.

There can be only one payload type per session. This has for consequences that multimedia transmissions have to be done using as many sessions as there are different media in a given transmission. This implies that there is no need to perform any media multiplexing operation on the sender's side and no media demultiplexing operation on the receiver's side. Also, it becomes easier to set up different quality of service reservations for each media to

transmit. For instance, if, during an audio/video transmission, the audio flow is considered to be more important than the video flow, we can reserve more bandwidth for the datagrams that have for destination the multicast address of the audio session.

During the existence of a RTP session, RTCP reports are sent periodically, by all the participants, to the session multicast address. The UDP port used for the RTCP messages is by convention the port next to the one used by the RTP transmission. RTCP reports are used to provide a transmission quality feedback to the session participants. With these information, they have the ability to adapt the transmission parameters such as, for example, the sampling quality of the transmitted audio data in an audio/video conference. The detection of losses over the network can also be used to perform such media quality scaling.

Each RTCP reports contains a *synchronization source identifier* (SSRC). Each SSRC inside a session, identifies a unique participant which is a data source. Along with the SSRC, the reports originating from a sender contain the number of datagrams sent since the beginning of the session, the octets quantity it constitutes, a RTP and a NTP[3] timestamp. The RTP timestamp corresponds to the time, relatively to the current session, when the current report has been sent. The NTP one corresponds to the absolute time when this report has been sent. With this information, it becomes possible to synchronize the presentation of different media into multiple sessions. The reports originating from a receiving participant contains, in addition to the SSRC, the highest received ADU sequence number, loss estimations, jitter measurements and timing information allowing to estimate the round trip time between the sender and the receiver.

Session participants also send *source description datagrams*. These datagrams contain additional information about the sending source, such as the canonical name of it (*CNAME*), which must be both human and machine readable and various personal information about the session participant.

RTCP traffic for one session must not exceed five percents of the total session traffic. Tunings can be done by the use of the information provided by the other participants.

2.3 Cryptographic concepts

In this section we expose the cryptographic concepts used throughout this paper. These concepts constitute the building blocks of the authentication

[3]Network Time Protocol

schemes described in the next chapter. We explain how public-key cryptography can be used to provide the authenticable and non-repudiable properties to the data exchanged between a pair of entities over an insecure network. The proof of correctness of the cryptosystems involved is not explained here since this is not the object of this study. This section is based on [20], [13] and on the specifications papers of the different standards that are presented.

2.3.1 Hash functions

Hash functions form the foundation elements of the schemes described in the next chapter. They are also used in the public-key based cryptosystems exposed into the next section. Their ideal goal is to provide a unique fingerprint for a given message. This fingerprint is intended to be used as an identifier for the considered message.

The unique argument of the hash functions we consider here and their results are byte sequences. Whatever the input message size is, the resulting fingerprints always have the same length.

Let h be a hash function, M a message and H the result of the application of h on M: $h(M) = H$. We have $\forall M : length(h(M)) = length(H) = l$ where $length$ is a function returning the length of the given byte sequence, and l is the fingerprint length for h.

A hash function must have additional properties to be considered sure and useful. First, it must be a one-way function. A one-way function is a function which is easy to apply but difficult to reverse. This means that the reverse computation of the hash function is not feasible with reasonable resources. So, we have $H = h(M)$ which is easy to compute, but $M = h^{-1}(H)$, where h^{-1} is the reverse function of h, which is not feasible with reasonable resources. It means that given M and its fingerprint H, it is difficult to find another message M' such that $M' \neq M$ and $h(M') = h(M)$.

h must also be collision resistant. It should be difficult to find two random messages M and M' such that $M \neq M'$ and $h(M) = h(M')$.

This last property is required to avoid, or at least to complicate, the birthdays attack. This attack is based on the following statistical observation; in a group of persons, in order to have at least two persons with the same birthday with a probability of a half, this group must be composed of at least 23 persons. This observation has for consequence for a hash function h, computing l bits long fingerprints, that an attacker would only have to test $2^{l/2}$ messages in order to have a good chance to find two of them with the same fingerprint. This can lead to message forgery when the original message is known from the attacker before its fingerprint is computed. A situation in which such a forgery can happen is given into [20]. Let's suppose a given

electronic contract which needs to be signed by two parties; Alice and Bob. As we'll see, the signature is applied on a fingerprint of the electronic contract, not on the contract itself. Now let's suppose that Alice wrote another contract in her favor. She could do some small imperceptible modifications (such as appending spaces or tabulations to the text) to both contracts in order to find a matching fingerprint between them. When this is done, she could present the original contract (containing the imperceptible modifications) to Bob for him to signed it (i.e. its fingerprint). Once the contract has been signed by the two parties, she can use Bob's signature along with her forged contract and pretend it was signed by Bob.

So, we see here that an important point is to choose a hash function which generates long enough fingerprints, depending on the potential resources the attacker can have access to. As we'll see, the most frequently used hash functions, generally offer fingerprint length of at least 128 bits. This means that in order to find two messages with the same fingerprint, an attacker would have to generate and test 2^{64} messages.

The two different hash functions used into the schemes implementation are MD5[4] and SHA1[5]. MD5 is the result of security improvements brought to the previous version MD4 which was designed to be simple, secure and fast. It is specified into [18]. It generates 128 bits long fingerprints.

SHA1 was designed to be used along with the american government signature standard DSA[6]. Its specifications can be found into [1] and [6]. It generates 160 bits long fingerprints.

Performance evaluations and comparisons of the Java implementation of those two hash functions are presented in chapter 5.

2.3.2 Public-key cryptography primitives

Public-key cryptography can be used to sign and authenticate data[7]. This process creates an unquestionable bond between an entity and the signed data. This has for consequence that the signing entity cannot deny having signed these data. Such a signature is non-repudiable. It allows to authenticate both the data and their source.

In public-key cryptosystems, each entity owns a key pair composed of a public key and a private key. This key pair is uniquely bound to the identity of the owning entity. The two different keys are used for different purposes. The private key can be used to sign data and thus, must never be disclosed.

[4]Message Digest version 5

[5]Secure Hash Alogrithm version 1

[6]Digital Signature Algorithm

[7]It can also be used for encryption purposes, but we do not make use of this feature.

The signature verification is done by using the public key of the signing entity. This key should be publicly available, since anyone should be able to check the validity of a signature made with the matching private key.

Two different public-key cryptosystems are available in the standard Java implementation: RSA and DSA. RSA[8] is based on the idea that factoring big numbers is a difficult problem. It is specified in [12]. DSA is based on the discrete logarithm computation problem. Its specifications can be found in [15]. We don't need to know how these systems are operating internally. We are only interested in how to use them.

Both key pairs in both cryptosystems are composed of a single bits sequence. The security level of these systems is a function of the length of the keys. For the implementation, we used 1024 bits long keys which is a minimum.

Both cryptosystems provide signature application and signature verification primitives. The signature application primitive makes use of the private key of the signing entity. In order to accelerate the processing, it operates on the fingerprint of the message which must be signed. It returns a bit sequence which is the signature. The signature verification primitive uses the public key of the signer and the message signature to operate. It returns a boolean value indicating a success or a failure, depending on the result of the signature verification.

Performance evaluations and comparisons of the Java implementation of those two cryptosystems are presented in chapter 5.

2.4 Constraints summary

We can conclude this chapter by expressing the highlighted key points.

Data flows over the Internet can be separated in two classes; the real-time flows and the non real-time flows. The difference lies in the fact that the content of the non real-time flows are entirely known before they are sent. This is not the case for the real-time flow as they are virtually infinite.

The requirements associated with real-time and non real-time streaming applications are different. In the real-time case, datagrams need to arrive on time to be accepted and treated. If a datagram is too late, it is discarded. This forces us to limit the time spent to treat the datagrams into the authentication process. For the non real-time flows this constraint has no sense.

The transmission is considered to be performed over a lossy and insecure network. Lossy means that some datagrams can get lost and so never reach

[8]Named by the initials of its designers: Rivest, Shamir and Adleman

the receiver. They could also be reordered or duplicated. These last two problems won't be addressed here. In practice, as we'll see, they are solved by the application layer on top of RTP. We'll only focus on dealing with the loss problem.

Insecure means that any entity present on the path of the datagrams we send can read them. This entity can also alter a flow by modifying the datagrams it contains and inserting its own datagrams in it. But we'll consider that no attacker can forge a public-key signature. They can just verify it, since the public-keys of the senders are known to everybody.

The authentication process must be highly loss tolerant. The majority of the remaining datagrams in an incomplete flow must be authenticable. We need to be able to authenticate each datagram independently of the others. A datagram will be considered either to be lost or authenticable. It must also be considered that different receivers for a given flow, will receive different datagram subsequences of this flow.

In order to minimize the network traffic between the sender and the receiver(s) of a flow, we need to limit the overhead added, by the authentication process, to the traveling datagrams. The time overhead introduced by this process should also be limited. This can be done by trying to use as less as possible cryptographic primitive applications by trying to amortize their use over multiple datagrams.

Chapter 3

Authentication and non-repudiation schemes

3.1 Introduction

In this chapter, we present different existing solutions to the authentication problem. We are interested in schemes which also offer non-repudiation. We highlight the mechanisms used to provide those two properties. At the end of this chapter, we compare the different techniques from a theoretical point of view. An empirical evaluation of them is presented in chapter 5.

3.2 Authentication and non-repudiation

In order to perform authentication we need a mean to bind the data we want to authenticate (i.e. the bare flow shown on figure 3.1 where the black arrow gives the direction of the flow) to something identifying a sending entity. The different schemes presented here do this by the use of public-key cryptosystems such as RSA and DSA.

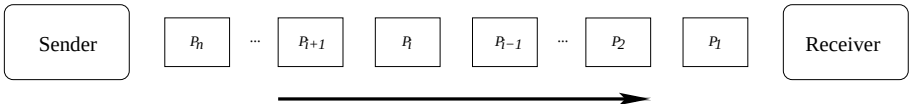

Figure 3.1: Flow without any authentication information

In public-key cryptosystems each entity has its own key-pair. The private part of a key-pair can be used to sign data in order to allow them to be authenticated by any other entity. The signed data is uniquely bound to

the entity who signed it and so, non-repudiation can be achieved. Indeed, this entity is the only one who can create authentication information which matches its public-key. Then, this entity cannot deny to be the sender of the signed data.

There are several ways to apply the signature process evoked above to a digital stream. These are described into the next sections.

3.3 Basic solution

A basic solution is to apply the signature primitive on each packet of the flow on the sender's side and then verify each individual signature on the receiver side (see figure 3.2). Once a packet signature has been tested by the receiver, two cases may appear: either the packet is authenticated and then it is passed to the application for its content to be processed, or the signature doesn't match and then the reception is halted. With this scheme, it is quite easy to prove to anybody both the authenticity of the source of the packet and the authenticity of its content since each packet has its own authentication information.

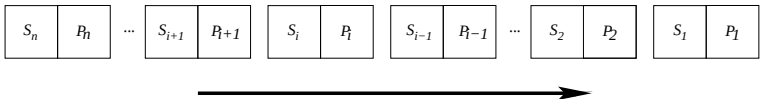

Figure 3.2: The basic solution

An advantage of this method is that there is no need to store more than one packet on both the sender's and the receiver's sides and so there are no requirements of additional buffers. For the sender, it means that the appropriate authentication information (i.e. the signature) can be appended to each packet without having to wait to know the content of some other packets. On the receiver's side, this means that every packet can be authenticated without having to wait to know the authentication information of some other packets (i.e. each packet can be authenticated independently).

The other advantage of this scheme is that it supports packet losses. As each packet can be signed and authenticated independently, the losses do not decrease the ability to perform either the signature or the verification. The signatures and verifications are always done in the same way, even if the loss rate is high.

The major drawback with this scheme is that there is a high computational load on both the sender and the receiver due to the intensive use

of public-key cryptography. Indeed, there is a signature application and a signature verification for each transmitted packet. A direct effect of this, is the introduction of an additional delay between packets. This may be unacceptable in systems where real-time transmission efficiency is a critical aspect.

The overhead associated with the signed packets depends exclusively on the signature algorithm which is used. The effect of each signature is spread over a unique packet. This leads to a high overhead on each of these.

See chapter 5 for overhead size and signature application time estimation.

3.4 Chain-based solution

3.4.1 Single-chained authentication

An other simple way of achieving our goal consists in chaining the packets of the flows. This is the first step into the idea of linking the packets together to reduce the number of signing and verification function applications. The following diagram illustrates the principles of simple chaining.

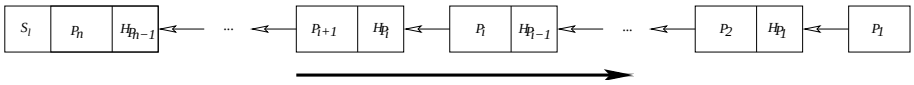

Figure 3.3: Single-chained authentication

Let's consider a stream. Let the payload of the consecutive packets composing this stream be P_1, P_2, P_3 ... and let h be a hash function. In order to achieve the packet chaining, we can embed the value of $h(P_1) = H_{P_1}$ into the next packet (i.e. packet number 2). So, this new packet will contain the payload of the initial packet number 2 (i.e. P_2) and a hash value of packet number 1. We can apply this for all the packets in the stream. So, as it is shown in figure 3.3, for all packet i, packet number i now contains H_{i-1}. The hash value of its payload is itself contained into packet number $i + 1$ and so on.

In the case of a finite stream, the result of a signature function applied on the last packet of the stream is then embedded into it. As we'll see, this is enough to provide authentication for all the packets of the stream. If the stream is virtually infinite, it can be split in small sequences of, for instance, a few hundred of packets, depending on the resources available on both the sender and the receiver. And then, the scheme can be applied to each sequence individually.

The verification is done step by step by verifying, for each packet i, the matching between the hash value contained in this packet (i.e. $H(P_{i-1})$) and the computed value $h(P_{i-1})$. This shows the validity of the packet number $i-1$, but not its authenticity. The authenticity is only proved when the signature of the last packet of the stream has been verified. Indeed, if the signature on the last packet (i.e. number n) passes the verification and if $h(P_{n-1})$ matches the hash value contained in packet number n (i.e. $H_{P_{n-1}}$), then the packet number $n-1$ is authenticated, and so is the whole stream.

This scheme allows to spread the effect of a single signature on several packets. The most direct effect of this, is the reduction of the number of public-key cryptographic operations performed. And so it reduces the computational load on both the sender and the receiver. The majority of the packets can be sent without the additional overhead of a signature.

On both sides, we only need to store the hash value of the previous packet to be able to process the current one.

The major drawback of this scheme is that there is no resistance to losses. If a single packet in a sequence is lost, the authentication chain is broken and so, the authenticity of the previously received packets cannot be established. The next schemes tries to avoid this kind of inconveniences by adding complexity to the linking between packets. Also, if the last packet of a sequence (i.e. the signed one) is lost, then the whole sequence is unauthenticable.

3.4.2 Multi-chained authentication

This scheme is presented in [16]. In this paper, it is called *EMSS* standing for Efficient Multi-chained Stream Signature. This scheme is designed to improve loss resistance of the previous one.

The stream is also split into sequences of a given length and the last packet of each sequence is also signed. The difference between this scheme and the previous one lies in the fact that a given packet is not only linked with its direct neighbors, but also with several preceding and following packets. The hash value linking technique is the same as the previous one.

In order to check if the authentication information contained into the incoming packets are valid, the receiver, first, needs to look if there exists a path composed of hash value links leading from the current packet to a verified signed packet.

The loss resistance is in general improved compared to the previous scheme, but it is still not as perfect as it is with the basic solution.

Figure 3.4 shows a flow on which this scheme has been applied. For the figure clarity, each packet only contains the hash values of the two following and the two preceding packets of the flow. The packets are represented in a

condensed way. For example, we have $P'_i = P_i + H_{i-1} + H_{i-2} + H_{i+2} + H_{i+1}$, $P'_{i+1} = P_{i+1} + H_i + H_{i-1} + H_{i+3} + H_{i+2}$, and so on, where H_i is the hash value of the payload of the packet i.

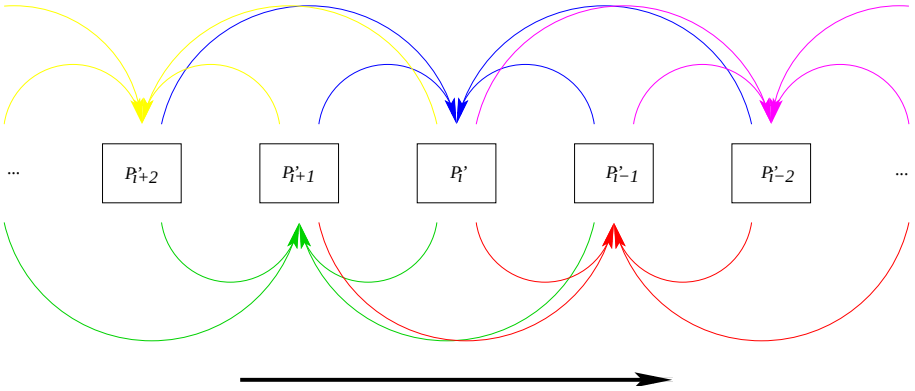

Figure 3.4: Multi-chained authentication

This scheme provides various degrees of loss resistance depending on the chosen embedded hash values. It appears that a randomly chosen selection gives the best results. But, for instance, a static selection like the five following and the five preceding packets is one of the worst alternative. This assertion comes from the simulation results exposed in [16]. Their simulation program showed that almost every selections are robust. It has been done by trying to estimate the verification probability for each packet of a simulated flow in relation to the length of the shortest hash value links path leading to the signed packet at the end of the current sequence. Figure 3.5, taken from [16], shows the simulation results for the comparison of a static section versus dynamic selection. The dynamic selection is clearly the best choice.

An improvement consists in the elimination of hash values redundancy. Indeed, the same hash value can be embedded several times into different packets. The solution is to split the hash values into a given number of chunks. All of these chunks are then distributed among the embedding chosen packets. A packet is considered as valid if we can find a certain number of its hash value chunks into the packets it is linked to. This improvement reduces the overhead on all the transmitted packets.

The same simulation as the one above has been led with the proposed improvement. Figure 3.6, taken from [16], shows the results. The bottom line is the same as the one of the dynamic selection in the static versus dynamic graph (figure 3.5). We can see that the elimination of the hashes redundancy globally improve the verification probability.

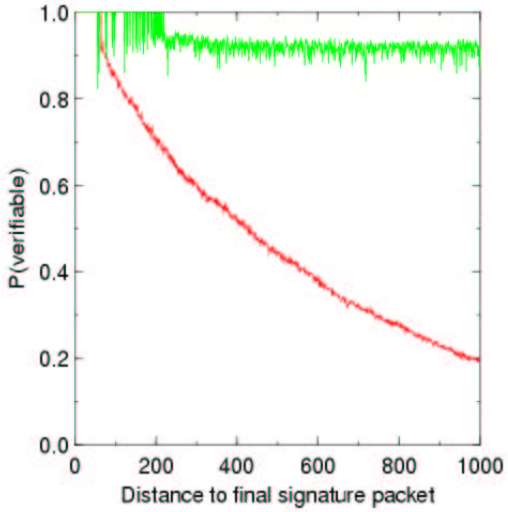

Figure 3.5: Verification probability for the dynamic selection (top line) and for a static selection (bottom line)

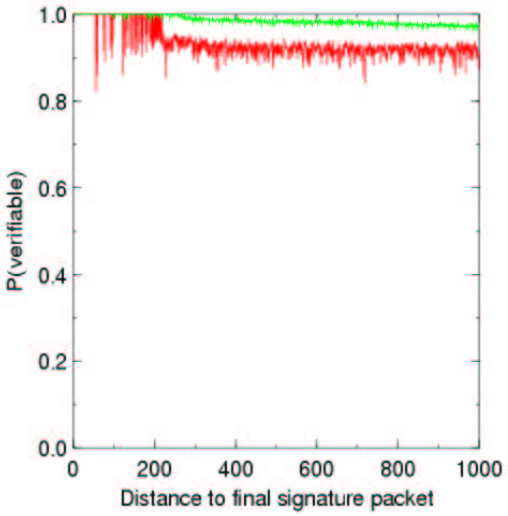

Figure 3.6: Verification probability for the solution with improvement (top line) and for the normal one (bottom line)

On the sender's side, there is a need to buffer the hash values (or more precisely the hash value chunks) in order to distribute them on the outgoing

packets. On the receiver's side too, but in order, here, to be able to check the validity of the incoming packets. The buffers size is a function of the maximum distance between the embedded hash values and the embedding packet.

There can be a delay between the reception of a packet and its authentication. This is due to the fact that we have to wait for a sufficient amount of verified hash values chunks to be reached. Once the fixed threshold is reach, the authentication process can take place.

If a signed packet is lost, the receiver is unable to authenticate the whole sequence.

3.5 Good solution characteristics

The previous solutions highlight the key characteristics a good solution has to provide. As we'll see in the next sections, a good solution often consists in a trade-off between all these parameters. Here they are:

1. Small signature delay;
2. Small verification delay;
3. Limited authentication information inter-dependencies;
4. Packet loss resistance;
5. Good signature amortizing ratio;
6. Small overhead.

3.5.1 Signature delay

This delay can be decomposed in two parts: the delay introduced by the *linking* phase and the delay introduced by the *signing* phase.

The linking phase is the phase during which the packets are linked together. This linking is done in such a way that the effect of a single signature (i.e. its authentication characteristic) is spread over several other packets. This allows to reduce the number of public-key function applications. This phase is not present in the basic solution, hence the delay it introduces is null. It is the main point of optimization which leads to the other solutions.

The signing phase is the phase during which the packet signature is computed and appended. The delay it introduces depends exclusively on the signature method which is used.

This delay must be kept as small as possible, since, the packets have to leave the sender as soon as possible.

3.5.2 Verification delay

In the basic solution, the verification delay is simply the time needed to apply the public-key signature verification function on the incoming packets.

With the introduction of the linking elements, the verification delay becomes twofold. It is composed of the verification time of the linking information[1] and of the verification time of the signature.

As the signing phase, this one should be kept as small as possible in order for the receiver to pass the packet as soon as possible to the next process stage of the application.

3.5.3 Authentication information inter-dependencies

With the basic solution, there is no dependencies between the packets. Each packet can be signed and verified independently, without having to wait for any other packet to arrive.

In the other solutions the repartition of the authentication information of a packet onto several other packets can be useful to raise its authenticability. But, by doing this, the receiver is forced to wait for several packets to arrive before being able to check the authenticity of a given packet. This significantly increases the verification delay.

The same observation can be made for the sender.

3.5.4 Packet loss resistance

In the basic solution, any packet can be lost without altering the ability of the receiver to perform verifications. This is not the case with all the schemes. For instance, the chain-based solutions are not fully resistant.

A good solution must be loss resistant since the considered network only offers best-effort traffic service. Any packet can get lost, and so the solutions must provide mechanisms or properties allowing to continue to authenticate arriving packets even on a high congestioned network.

3.5.5 Signature amortizing ratio

The signature amortizing ratio is the measurement of the linking degree offered by a solution. This is for a given scheme, the ratio between a certain

[1] The verification of the packet links is called the validity check.

number of signatures and the number of packets that can be authenticated by the use of these signatures.

The smaller this ratio is, the best the signature amortizing is. But, the greater this ratio is, the best the loss resistance of the considered scheme is. This assumption is especially valid for the single chain solution, since the linking is quite weak. If a single packet is lost, the whole chain becomes unauthenticable.

3.5.6 Overhead

The overhead is the mean size of the additional information appended or embedded into the packets of a stream to provide authentication. It is composed of the linking information and of the signature if there is one.

Overhead has an impact over the efficiency of the transmission. The smaller the overhead is, the smaller the transmission delay is.

3.6 Tree-based solution

This solution is taken from [21]. At first, they present the star- chaining idea and then, they generalize it to the tree-chaining solution. The principle of this solution is based on the idea of splitting the flow into several blocks of consecutive packets. The authentication process is then applied to each of these blocks. This allows to perform a single signing (verification) operation for a group of packets on the sender (receiver) side.

3.6.1 Star-chaining

Let's consider a flow split into blocks of m packets. A block can be represented by $(P_1, P_2, \ldots, P_m)$, where P_1 is the payload of the first packet of the block and so on.

Let $H_1, H_2, \ldots, H_m$ be the results of h applied to the payload of the packets of a given block. For example: $h(P_1) = H_1$.

The *block hash value* H_{1-m} is the result of h applied to the concatenation of $H_1, H_2, \ldots, H_m$: $H_{1-m} = h(H_1 \bullet H_2 \bullet \ldots \bullet H_m)$ where $\bullet$ is the concatenation operator.

The *block signature* $S(H_{1-m})$ is the result of the application of a public-key signature function on the block hash value $sign(H_{1-m})$ where $sign$ is this public-key signature function.

We can define the signature associated to a packet as the concatenation of the signature of the block it belongs to, its position in this block and the

hash values of all the other packets present in the block. This is called the *packet signature*.

The verification of an incoming packet on the receiver's side consists in the reconstruction of the block hash value and on its signature verification. The reconstruction is done by computing the hash value of the packet with the h function and inserting it into the hash value sequence by concatenation. So if P_i is the payload of the incoming packet and $H'_i = h(P_i)$, then the block hash value is $H'_{1-m} = h(H_1 \bullet H_2 \bullet \ldots \bullet H'_i \bullet \ldots \bullet H_m)$, where $(H_1, H_2, \ldots, H_{i-1}, H_{i+1}, \ldots, H_m)$ is found into the packet signature. The signature verification is then performed by a public-key signature verification function. The packet is considered to be valid if, and only if, $sign(H'_{1-m}) = S(H_{1-m})$, where $S(H_{1-m})$ is known since it is also present in the packet signature.

Figure 3.7 illustrates the example of the processing of an incoming packet on the receiver's side. Let's consider the arrival of packet number 3 of a given block. This is the first arrived packet belonging to this block. This packet contains the payload of the original packet coming from the flow before the application of the authentication scheme, the signature of the block it belongs to, its position in this block and the hash values of all the other packets belonging to this block.

The first step into the verification of the authenticity of this packet consists in the reconstruction of the block hash value. In order to accomplish this, we first have to compute the hash value H_3 of the original payload (i.e. the payload of the packet as it was before the application of the authentication scheme to the flow) of the packet. Next, we just have to insert the computed hash value into the hash value sequence (i.e. $H_1, H_2, H_4, H_5, H_6, H_7$ and H_8) contained into the packet using its position (i.e. 3) which is also contained in the packet. Once we have the complete hash values sequence, we can concatenate all its components together and then compute the hash value of the result. This is the block hash value. The final step is the verification of the signature contained into the packet in relation to the computed block hash value. If the signature verification has failed, then it means that the packet has been altered during the transmission and that it is not authenticated. If the signature is verified, the current packet is considered as to be authenticated. The hash values of the other packets of the current block can be stored in a cache in order to accelerate their future authentication. Indeed, as the signature verification is successful for the first arrived packet, we only have to check if the computed hash values of each other packets of this block match the ones stored into the cache. This is enough to authenticate them.

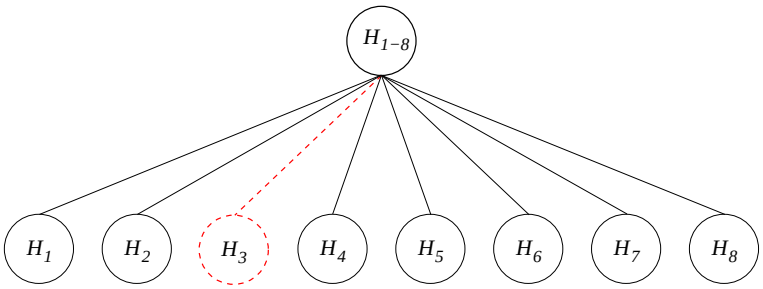

Figure 3.7: Authentication star, example

3.6.2 Tree-chaining

The tree-chaining technique is a generalization of the star-chaining idea. In this case, the packet hash values of a given block are considered to be the leaves of a rooted tree. This rooted tree can be of any degree. Only the root of the tree is signed, this results in the block signature.

Apart from the leaves, any node of the tree is the result of the application of the h function on the concatenation of its sons. For example: $H_{1-2} = h(H_1, H_2)$, $H_{1-4} = h(H_{1-2}, H_{3-4})$, ...

The packet signature can be decomposed into the block signature, the position of the packet in its block and the siblings of each node present on the path leading from the packet to the root of the tree.

With this method, the length of the signature associated with a packet can be significantly reduced. Indeed, with star-chaining, the number of hash values present in the packet signature was $m - 1$ hash values long, it is $(log_{deg}(m)) * (deg - 1)$ hash values long with tree-chaining, where deg is the tree degree. This is the main difference between those two schemes.

The sender needs to buffer as many packets as there are in a block before it can start the application of the scheme. This is why, the tree building phase on the sender's side can be time expensive and so, this scheme can be inappropriate for real-time stream processing.

When a packet arrives at the receiver, the verification is performed by testing the validity of each node present on the path leading from the packet hash value to the root of the tree. Then the signature of the root is tested.

To verify a node, we just need to be sure that its father is the result of the application of the h function on the concatenation of all its sons. This can be done by computing the hash value of the arrived packet, concatenating it with its siblings and then applying h to the resulting sequence. The siblings of every node on the path can be found into the packet signature.

If all the nodes present on the path to the root are valid, the root signature

can be checked. This is done in the same way as it was in the star-chaining solution.

There is no delay between the reception of a packet and the beginning of its authentication. The linking validation phase is not necessary since the authentication can be done on each packet independently. All the necessary information is contained in each incoming packet.

Figure 3.8 illustrates an example of authentication on the receiver's side. Let's consider the incoming packet number 3. This is the first arrived packet belonging to this block. In order to authenticate it, we have to rebuild the hash value of the current block and verify the signature present in the current packet. To do this we have to check the nodes present on the path leading from the hash value of the arrived packet to authentication tree root (i.e. the block hash value). These are the nodes in red in figure 3.8. So, the first step is to compute the hash value H_3 of the payload of the original packet (i.e. the payload of the packet before the application of the authentication scheme). Next, we have to compute the hash value of $H_3 \bullet H_4$ where H_4 is the hash value of the original packet number 4. It is contained in the packet number 3 since this is a sibling of the node H_3 into the authentication tree. Next, we have to compute H_{1-4}. This can be done by using the hash value of $H_{1-2} \bullet H_{3-4}$ where H_{1-2} is a sibling of the node H_{3-4} into the authentication tree. It can be found, like H_4, in the arrived packet. Now we are able to rebuild the block hash value. It can be done by computing the hash value of the concatenation of H_{1-4}, which we just computed, and H_{5-8}, which can be found into the current packet. Indeed, H_{5-8} is also the sibling of a node present on the path going from the current packet to the root. Once the block hash value is known, we can verify the signature contained in the current packet. If the signature appears to be invalid, then the packet is considered as not being authenticated. If the signature verification is successful, the current packet is considered as to be authenticated. At this point, we can store the red and green nodes (3.8) of the authentication tree along with its root. Indeed, let's considered the next arriving packet be the number 4 of the current block. In order to authenticate it, we only have to compute its hash value and check if it is same as the one we stored into the cache. If this verification is successful, then the packet number 4 is authenticated.

Now let's examine what happens when packet number 5 arrives. The blue node in figure 3.9 are the one stored into the cache. In order to authenticate it, we have to compute H_{5-6} from H_5 and H_6 in the same way as it was done for H_{3-4} with H_3 and H_4 when the packet number 3 arrived. We can also compute H_{5-8} from H_{5-6} and H_{6-7} in the same way. Now, we are able to compute the block hash value H_{1-8}. As H_{1-4} and the block hash value H_{1-8} are already in the cache, to authenticate the current packet, we only have

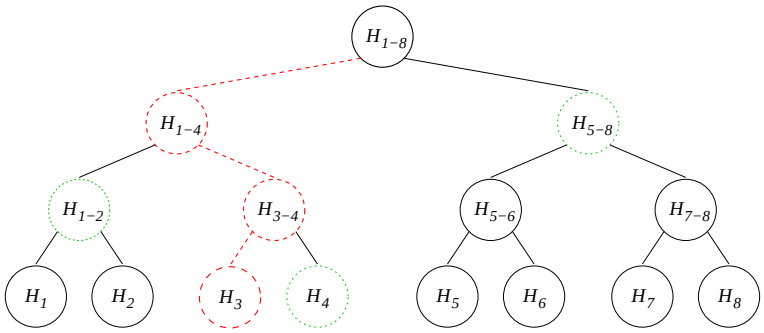

Figure 3.8: Authentication tree, first example

to compare if the computed block hash value matches the one stored in the cache.

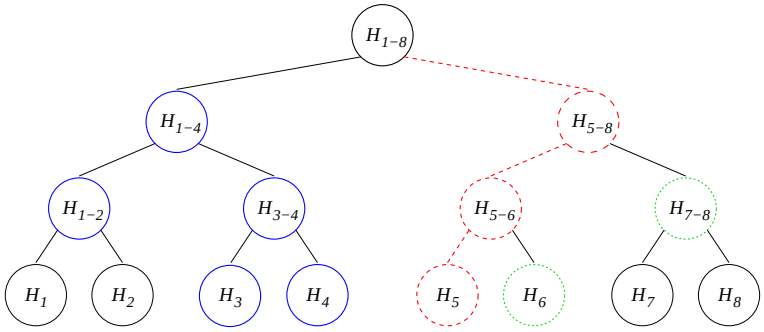

Figure 3.9: Authentication tree, second example

3.7 Graph-based solution

The graph-based solution is also an evolution of the chain authentication schemes. It is presented in [8] and [14]. This scheme constitutes a generalization of all the chaining schemes seen before.

This solution tends to adapt the hash value linking structure in order for the stream authentication to resist to bursty losses like they appear on the Internet. This adaptation is done by trying to increase the probability of authentication for all the packets.

An authentication graph is an oriented graph where the nodes can be seen as the packets and the edges as the hash value links. The orientation of the edges is going from a packet to the packet containing its hash value.

The idea behind this scheme is for the sender and the receiver to agree on an authentication graph.

On the sender's side, the authentication phase consists in building the authentication graph. This phase can require to buffer some packets in order to wait for them to be included in the graph. The receiver can also have to buffer some packets before it can authenticate them. This buffer need depends on the authentication graph used.

The difficulty with this scheme is to find a good authentication graph allowing to deal with the different trade-offs such as authentication delay and buffer size.

3.8 Schemes comparison table

	Signature delay	Verification delay	Packet inter-dependencies	Loss resistance	Signature amortizing ratio	Overhead
Basic solution	1 sig	1 sig	None	100%	1	1 sig
Single-chain	1 hash + (1 sig / seqSize)	1 hash + (1 sig / seqSize)	1 incoming and 1 outgoing edges	100% if seqSize $==1$ 0% if seqSize $\to \infty$	1/seqSize	1 hash + (1 sig / seqSize)
Multi-chain	m hash + (1 sig / seqSize)	m hash + (1 sig / seqSize)	m incoming and m outgoing edges	100% if seqSize $==1$ 0% if seqSize $\to \infty$	1/seqSize	m hash + (1 sig / seqSize)
Star-chaining	(1 sig +(seqSize +1) hash) / seqSize	1 sig +2 hash	None	100%	1/seqSize	sig + (seqSize - 1) hash
Tree-chaining	(1 sig + n hash) / seqSize	1 sig +(log_{deg}(seqSize) +1) hash	None	100%	1/seqSize	(log_{deg}(seqSize)) $*(deg-1)$
Graph solution	Depends on the graph	Depends on the graph	Depends on the graph	100% if seqSize $==1$ 0% if seqSize $\to \infty$	j/seqSize	Depends on the graph

Table 3.1: Scheme comparison table

Table 3.1 presents a comparison of the different schemes presented in this chapter. This comparison relates to the key characteristics described into section 3.5.

In all the table, *seqSize* represents the length of the sequences treated by the schemes. In the signature and verification delay columns, *sig* and *hash* represent the time spent to apply, respectively, a single signature and to compute a single hash value. In the Overhead column, *sig* and *hash* represents the size in bytes of respectively a single signature and a single hash value.

In the tree-chaining line, *deg* represents the degree of the considered tree. n is a constant giving the number of node of the considered tree. The formulas given for star and tree-chaining do not consider the possibility of caching the computed values. In the multi-chain line, m is the constant giving the number of hash values which are contained into a single packet. The used of hash value chunks instead of entire hash values is not considered here. For the graph solution, j is a variable giving the number of signatures which spread their effetcs on a single packet sequence.

Part II

Implementation

Chapter 4

Application presentation

4.1 Introduction

In this chapter we present the prototype we developed in the scope of the IST[1] project *ShopAware*[2].

Along with the electronic commerce development lies the idea of electronic contract. Electronic contract signatures can be performed by using public-key cryptography and can be considered as an equivalent to handwritten signature for classic contracts if they are executed with respect to a few basic principles. Electronic contracts can contain more than plain text. For example, any multimedia information contained in files could be appended to any existing electronic contract. The only restriction is that these additional information (multimedia or of any other type) must be known in advance and come from a trusted source.

A problem appears when, for instance, the electronic contract has to contain audio/video conference recordings done together with the other(s) contractor(s) over a lossy and insecure network such as the Internet. Indeed, interactive audio/video conference is one application field for real-time flows. In this situation, the recorded data cannot be blindly trusted. In absence of appropriate mechanisms such as those described in the previous chapter, there is no warranty over the integrity of the received data and over the source identity of these data. Furthermore, there is no mean to avoid repudiation of the recordings by one of the contractor.

Our prototype implements some of the schemes defined in chapter 3 and thus, holds the basic capabilities of an application which allows non-repudiable communication recordings to be performed from a real-time mul-

[1] See `http://www.cordis.lu/ist/`

[2] Project Reference: IST-1999-12361

timedia communication between two entities over a given network.

The whole prototype is written in Java[3]. This language has been chosen for interoperability reasons. Indeed, the other parts of the ShopAware project were already written in Java at the time we started the prototype development. We also use the Java Media Framework (JMF) application programming interface[4] (API). This API provides the necessary classes to manipulate time-based media such as audio and video and to transmit these media with RTP over a network. It incorporates a plugin architecture allowing to write custom media processing routines.

Our implementation is limited to the transmission of H263 video, no sound capabilities have been implemented. Actually, this is not a fixed limitation. H263 was chosen because this is one of the only video standards which can be coded and decoded by JMF in real-time. Any other media could be transported by our prototype. This would require minimal changes of some parts of the code. Our prototype was built with extensibility in mind.

4.2 Global architecture

Our prototype is composed of two main programs: a client and a proxy server, and two utilities: a basic public key pair generator and a recording checker.

Each communicating entity has to run a client and must be able to reach a proxy server by a trusted network. By trusted, we only mean that no one will try to tamper with the passing datagrams over this network. There is no restriction over the loss rate of the different networks, except that it should not exceed the tolerance level of an ordinary audio/video conference application. This condition is only stated in order to obtain significant results.

The figure 4.1 shows a possible configuration of the system. The white arrows represent bi-directional communications without authentication information. The black arrows represent bi-directional communications containing additional authentication information. In this example, the client #1 is communicating with the client #4 via the proxy servers #1 and #3. The client #2 and #3 are communicating together through the proxy servers #2 and #3. Note that a proxy server can treat several communications at the same time, this is the case here for the proxy server #3.

A communication is always initiated by one of the two clients, it is called the *source* client. The other one must be in a waiting state. The communication initialization is performed by contacting a proxy server via a trusted

[3]See `http://java.sun.com/`

[4]See `http://java.sun.com/products/java-media/jmf/index.html`

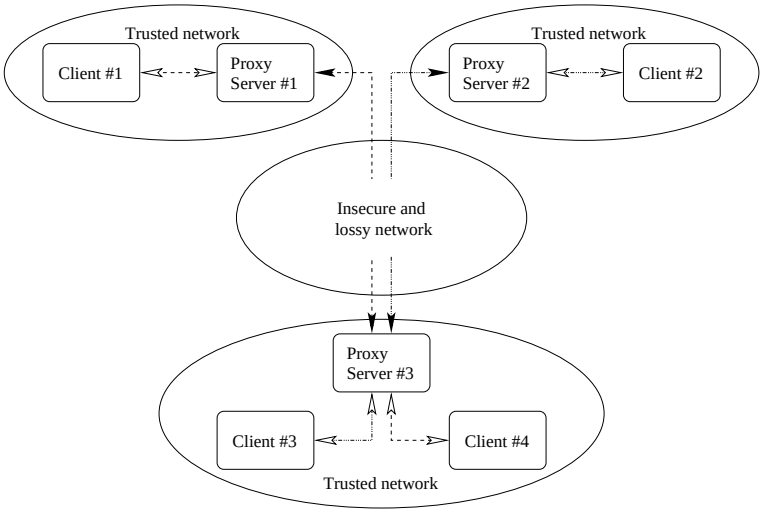

Figure 4.1: Prototype architecture possible configuration

network. This proxy server is called the *source* proxy. Once the source proxy has been contacted, the source client sends him the following communication parameters:

- The address of the proxy server (called the *destination* proxy) used to reach a given client (called the *destination* client);
- The address of the destination client;
- The requested communication type.

The requested communication type parameter tells the proxies and the destination client what will be the direction of the communication. It can be either bi-directional, from the source client to the destination client or from the destination client to the source client. Note that the machine from which a video stream originates must be equipped with an appropriate video capture device compatible with the JMF layer[5].

Before the beginning of a communication, each client has to generate a DSA key pair. This can be done once for all by using the key pair generator. The source and destination private keys must be accessible respectively for the source and destination proxy. And the source and destination client

[5]A list of capture devices known to be working under Windows and Solaris environments can be found at `http://java.sun.com/products/java-media/jmf/2.1.1/formats.html\#Capturers`. Under the Linux environment, the capture devices, for which a device driver built on top of the video4linux architecture is available, should also work.

public keys must be accessible respectively for the destination and source proxy. This is shown in figure 4.2.

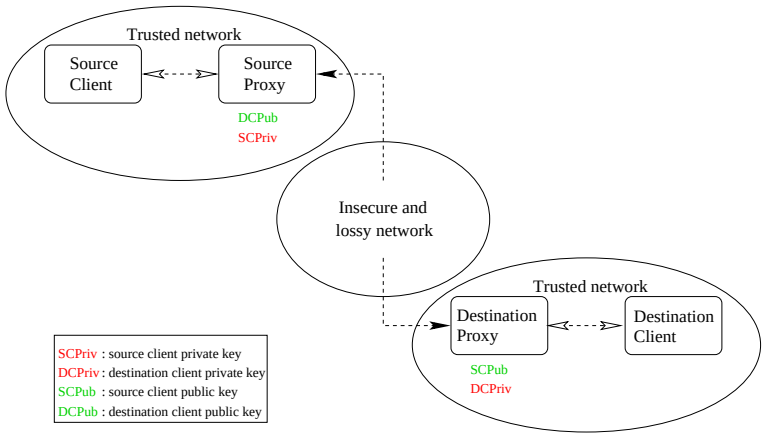

Figure 4.2: Prototype key distribution

Once the source proxy has been contacted and the communication parameters have been sent to him, it will try to create a connection with the destination proxy in order to send him the address of the destination client and the communication type requested by the source client. When the destination client is ready, an acknowledgment is sent back from the destination client to the source client through the destination and source proxies. Then the communication can begin.

During the communication, the real-time flows involved are recorded as files on each proxy performing reception. Their authenticity can be checked with the recording checker utility. In order to do this, the public-key of the entity who signed the recorded flow must be available to the entity who performs the check.

4.3 Internals

In this section we expose the internal functioning of our prototype. We start by presenting the API which is used. Next, we describe the internal architecture of the different programs we developed.

4.3.1 Java Media Framework

This section is based on [2], [5] and of the study of some pieces of example code. We developed the prototype at the end of 2001. At this time, the

only accessible documentation was the JMF programmer's guide written in November 1999 (see [2]) and the code samples[6] provided by Sun Microsystems for demonstration purposes. It appears that the JMF programmer's guide wasn't matching the JMF API implementation anymore, and this, especially for the RTP transmission API part (which is the most interesting one).

JMF can be used to render, process and transmit time based multimedia data. These data can come from files, capture devices or RTP sessions and be destined to files, rendering devices or RTP sessions. A plugin architecture allows to apply modifications to multimedia data.

JMF API is mainly composed of Java interfaces. Objects implementing these interfaces can be obtained through the use of a set of intermediary objects called the managers. All the resources they provide are system dependent. There exists four of such managers which are all implemented by a public final class containing only static methods and variables. These classes are:

- `Manager`;
- `PlugInManager`;
- `CaptureDeviceManager`;
- `PackageManager`.

The `Manager` class provides static methods allowing to create objects such as players and processors which are the main engines of JMF based applications. The `PlugInManager` class gives information about the registered plugins and allows to register custom plugins into the system. The `CaptureDeviceManager` offers the same kind of services as the `PlugInManager` class does, except that it is relative to the available capture devices on the system. Finally, the `PackageManager` class allows to maintain a registry of all the JMF package extensions available on the system. For instance, a custom implementation of some JMF interfaces can be registered into the system through the static methods of this class.

Let's examine how players and processors are created and operates. Here is the definition[7] of what a player is in JMF terms:

> A player is an object used to control and render multimedia data that is specific to the content type of the data.

[6] See `http://java.sun.com/products/java-media/jmf/2.1.1/solutions/index.html`

[7] This definition comes from the JMF API documentation available at `http://java.sun.com/products/java-media/jmf/2.1.1/apidocs/`

It means that all players are different, depending on the data type they have to render. This information is passed at the creation of a `Player` object. Indeed, in order to create a player, we must provide a `DataSource` object. A `DataSource` object can be seen as the abstraction of a data container, it encapsulates a data stream. It is used to retrieve information about the type of the data it contains and to have access to these data.

Once a `Player` has been created using a given `DataSource`, it must pass through several states before it can start rendering the media. Figure 4.3[8] shows a graph representing the transition relations between the possible states of a `Player`. Just after the instantiation of a new `Player`, this `Player` is in the unrealized state. At this point, this `Player` doesn't known anything about the media it has to render. Once the `realize()` method has been called, the player enters the realizing state. During this phase, the `Player` retrieves information about the media it has to render and tries to acquire non-exclusive resources such as the one used to render this media. When the `Player` reaches the realized state, it knows what kind of resources will be needed to render the associated media. At this point, it is also possible to retrieve all the available visual components allowing to view and control this media. The prefetching state, reached after a call to the `prefetch()` method, is the state during which the `Player` performs the last preparations before the beginning of the presentation. It consists in the acquisition of exclusive resources such as, for instance, the access to a capture device from which the media data originates. The prefetching state is also the state where the `Player` begins to load media data from the `DataSource`. Once the `Player` reaches the prefetched state, the presentation can be started by a call to the `start()` method. If, for example, the user uses the visual control components (if it is available) to seek into the presented media, the `Player` will have to go back into the realized state and recall the `prefetch()` method before it can continue to present the media. Changing the current presentation position into a media flow, is done by a call to the `setMediaTime()` method. A call to the `stop()` method when the `Player` is in the started state, brings it back to the prefetched state. Both players and processors implement the JMF `Controller` interface. This interface defines the different methods used to switch between states.

When a `Player` is started, it must refer to system time in order to synchronize the media presentation. The time model used by JMF to do this is quite simple. For each data flow, JMF keeps a time counter indicating the current time position in the flow. In order to be able to synchronize the data presentation with wall-clock time, JMF also maintains an independent time

[8] This figure comes from [2].

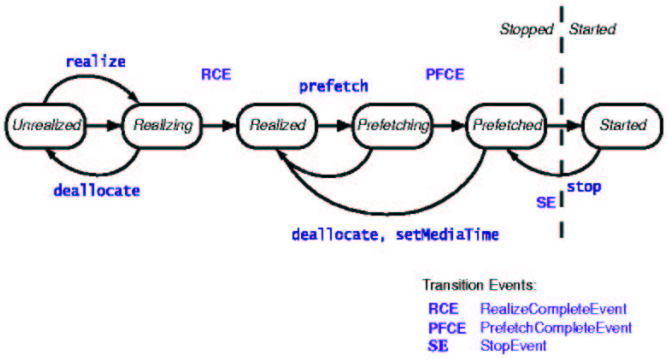

Figure 4.3: Player states graph

counter called the time base. This counter is based on the internal system clock and provides periodical references in time. It cannot be stopped or reset. During the presentation of a media, JMF constantly maps the time counter associated with the data flow with the time base. Operations such as those having an effect on the sequencing of the presentation (i.e. stopping, rewinding, ...) are performed on the time counter associated with the flow, not on the time base. In this way, a rate factor can be easily applied on the speed of the presentation. For example, in order to double the speed of the presentation, JMF just doubles the speed of the passing time for the counter associated with the flow. This remark on changing the speed presentation of a media is only valid for media contained in files. Indeed, it is not possible to know in advance what will come out of a capture device or from a RTP session.

During its existence, a `Player` generates some events to inform the other parts of the application about its state. These events are implemented by the class `ControllerEvent` and can be detected by the installation of a `ControllerListener` onto the `Player` (which, as we've already said, implements the `Controller` interface) by a call to the `addControllerListener(ControllerListener listener)` method. The `ControllerListener` must implement the void `controllerUpdate(ControllerEvent event)` method which is called each time a `ControllerEvent` is posted by the listened `Controller`. The transition events posted by a `Player` are `PrefetchCompleteEvent`, `RealizeCompleteEvent`, `StartEvent` and `StopEvent`. They are all used to indicate that the `Player` has reached a given state.

Processors are different from players. They are not intended to render media, but to apply them some treatments and output the result to various destinations such as files or RTP sessions. Processors are created in the same

way as players are, by providing a `DataSource` object to a static method belonging to the `Manager` final object.

The states graph of a processor is slightly different from the one of a player. The states graph of a processor contains two more states situated between the unrealized and realizing states. These are the configuring and configured state which can be reached from the unrealized state by a call to the `configure()` method. When a processor reaches the configuring state, it tries to retrieve all the information about the media it has to treat from the `DataSource`. This is the only state when the output format of the processor can be set. This is done by a call to the `setContentDescriptor(ContentDescriptor outputContentDescriptor)` method. The other states and the transitions between them are the same as in the case of a player (see figure 4.4 for details). There is an additional `ControllerEvent`, the `ConfigureCompleteEvent` which is posted when the configured state is reached.

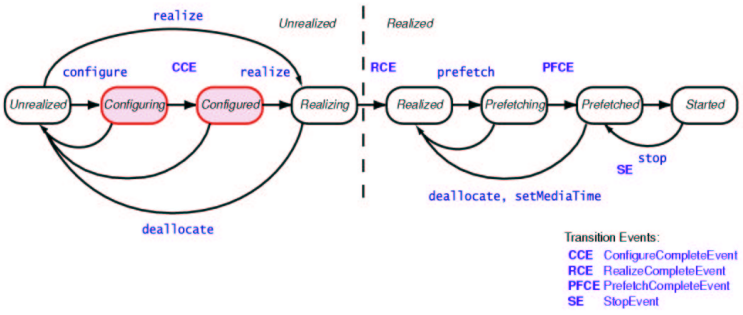

Figure 4.4: Processor states graph

Players and processors can apply modifications to a media data flow passing through them. This can be done by the utilization of plugins. There exist five types of plugins: codecs, renderers, multiplexers, demultiplexers and effects. Codecs are used to encode or decode media data. This can be done, for instance, as a preprocessing step before the presentation of the media on screen or in order to convert it into another media format. Renderers are used to present the media on the rendering device such as a screen. They constitute, for example, the interfaces between JMF and the graphical user environment such as X Window system. Multiplexers are used to mix multiple media flows into a single one. This is the kind of plugin which is used, for instance, in order to allow the recording of both audio and video flows into a single file. Demultiplexers do the opposite job. They are used to separate multiple media flows contained in a single one before applying

individual treatments to each of them. And finally, effects are used to apply changes directly to a media contained in a flow. For instance, this can consist into the application of a volume increment to the audio data contained in a flow. Plugins can also be organized in chains in order to apply several consecutive modifications to a flow.

In order to be usable, a plugin must be registered into the system. This can be done by using an intermediary manager, the `PlugInManager`. This manager provides several static methods allowing to add or remove plugins from the plugin database and to get information about them. All these mechanisms allow to build custom plugins. In order to be recognized and usable, a custom plugin must implement one of the five interfaces corresponding to the five plugin types evoked above. Each plugin, whatever its type, has an input and an output format which characterizes the media type on which this plugin is applicable.

RTP transmission is the most difficult part to understand in the JMF API. And this, because of the lack of up-to-date documentation. Indeed, as we've already said, the JMF API doesn't match the explanations given into the JMF programmer's guide[9] anymore. The rest of this section is based on sample code found on the Sun Microsystems website.

In order to transmit RTP data with JMF, a `SendStream` object connected to the data output of a `Processor` must be created and started. The creation of such an object can be done by calling the `createSendStream()` method of an initialized `RTPManager`. The creation of a new `RTPManager` is done by calling the `newInstance()` method of the static object `RTPManager`. Its initialization can then be performed by calling its `initialize` method with the RTP session parameters.

The reception of a RTP flow is done by listening for incoming flow on a local representation of the considered RTP session. A local representation of a RTP session posts a `RTPEvent` object each time a noticeable event occurs on it. More particularly, a `NewReceiveStreamEvent` object is posted each time a new flow is received. A `ReceiveStream` object can be retrieved from this event by a call to the `getReceiveStream()` method. Then a call to the `getDataSource()` method returns to associated data source which can be connected to a processor.

4.3.2 Transmission parameters negotiation

Each program constituting our prototype is composed of several modules which need to be fed with some parameters values before they can be used.

[9] See [2]

This is mainly the case for the modules used to manage the RTP sessions. These modules need to known where to send data and where to look for incoming data flows.

As already been said in section 4.2, a client can be either a source client or a destination client. A source client is intended to be in direct relation with a source proxy. A communication is always initiated by a source client.

When invoked from the command line, some parameters are passed to a source client. These parameters were already rapidly evoked in section 4.2. Here is how they are used. Seven values must be passed to a source client when it is invoked:

1. A source proxy IP address identifying a network interface of a machine on which a proxy is waiting for connections;

2. The source proxy TCP port on which it listens for communication initiations;

3. A destination proxy IP address identifying a network interface of a machine on which a proxy is waiting for connections;

4. The destination proxy TCP port on which it listens for communication initiations;

5. A destination client IP address identifying a network interface of a machine on which a client is waiting for a connection;

6. The destination client TCP port on which it listens for communication initiations;

7. The requested communication type which can be either the `rx`, `tx` or `rxtx` tokens, depending on the source client capabilities. It corresponds respectively, from the source client point of view, to reception only, transmission only and reception/transmission all together.

Figure 4.5 shows the different steps of the initialization of a source client. When the client program is launched as a source client (with seven command line parameters), it starts in state number 0. At this point, the source client tries to send the five last parameters it received to the specified source proxy on the given TCP port along with a type identifier (i.e. the *client* token in this case). When this is accomplished, the source client reaches the state number 1 and waits until a response comes from the source proxy. If the received answer is the *error* token, then the source client reaches state number 4 and is halted. This can happen if a problem occurs on one of the other parts of the

system (i.e. the source proxy, the destination proxy or the destination client). In case of success, the received answer is composed of a `session ID` and of a `RTP transmission port` and/or a `RTP reception port` depending on the requested communication type. The `session ID` is used, on the clients, to allow them to identify the files containing the records of the passed communication. The `RTP reception port` and the `RTP transmission port` are respectively used to configure the client parts which manage the RTP reception and transmission (See section 4.3.3 for details). If an error occurs during the configuration of these managers, the *error* token is sent to the source proxy and the source client reaches state number 4 and is halted. This can happen, for example, if the selected capture device is not available at the time the source client asks the `CaptureDeviceManager` to use it. If the source client is ready to start the communication, it sends the `ok` token to the source proxy and reaches state number 3. Once it is done, the source client starts its `ClientRTPTransmitter` and/or its `ClientRTPReceiver` thread(s) (See section 4.3.3 for details) depending on the requested communication type.

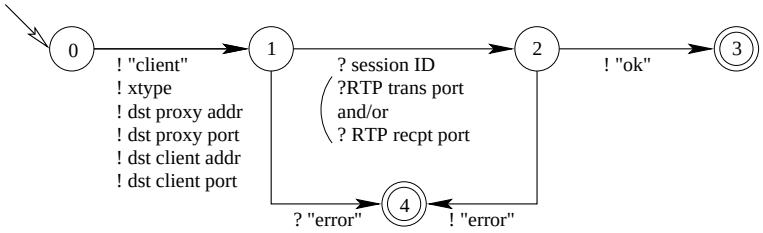

Figure 4.5: Source client finite state machine used during the initialization phase

The initialization phase of the proxies is a bit more complex since they have to exchange parameters values with a client and another proxy at the same time. A proxy is started with for command line arguments, the TCP port on which it listens for communication initiation and the maximum number of communications it will accept at the same time. A proxy can handle several communications at the same time. When it is started, a proxy immediately enters a waiting state. At this point, it can't be considered as a source or as a destination proxy. Indeed, it only depends on the type of the entity which establishes a connection with it and at this point, it is not known. The first received data will determine the proxy type. In the case of a source client trying to initiate a communication, the first received data contain the *client* token. In the case of a source proxy trying to propagate

the communication attempt of a source client, the data contain the *Proxy* token.

Figure 4.6 shows, from a source proxy point of view, what happens when a source client tries to initiate a communication. Note that all exchanged messages on figure 4.6 are prefixed with either a *d* or a *s*. This indicates that the considered message originates from or is destined to a destination or source entity.

When waiting for connection, the proxy is in state number 0. After the successful connection of a source client, it receives the *client* token, the value of the parameter describing the requested communication type (xtype), the IP address and TCP port needed to reach a destination proxy and the IP address and TCP port needed to reach a destination client. Now, we know that the proxy is a source proxy. After the reception of the data, it jumps to state number 1. At this time, it attributes a session identifier to the communication that is being initiated. Then it tries to send the communication parameters to the identified destination proxy. These parameters are the *proxy* token, the `session ID`, the requested communication type, the IP address and TCP port needed to reach a destination client and finally the `RTP reception port` and/or the `RTP transmission port`. The `RTP transmission port` indicates what UDP port is used for the RTP session having for sending participant the source proxy and for receiving participant, the destination proxy. The `RTP reception port` indicates what UDP port is used for the RTP session having for sending participant the destination proxy and for receiving participant the source proxy. Both `RTP reception port` and `RTP transmission port` are attributed by the source proxy. At this point, if any error occurs on the source proxy it sends the *error* token to the source client. In this case, the source proxy goes into state number 9 and the current communication establishment attempt is simply discarded. When the parameters have been successfully transfered to the destination proxy, the source proxy goes into state number 2 and wait for the answer of the destination proxy. This answer can either be the *ack* token or the *error* token. In this last situation (state number 7), the source proxy sends an *error* token to the source client and the current communication attempt is discarded (state number 9). But, if the *ack* token is received, then the source proxy attributes one or two new UDP port(s) (depending on the requested communication type) for its RTP communication with the source client. If a problem occurs on the source proxy, then an *error* token is sent to the destination proxy (state number 9). The current communication attempt is then discarded. If no problem occurs, then it sends the `session ID` along with the attributed RTP session port(s) to the source client (state number 4). If the source client answers with an *error* token (state number 8), then the source

proxy sends the same token to the destination proxy (state number 9). Then, the current communication attempt is discarded. If the source client answers with an *ok* token (state number 5), it is forwarded to the destination proxy. Finally (state number 6), the source proxy starts its RTP session managers according to what has been configured in the previous steps.

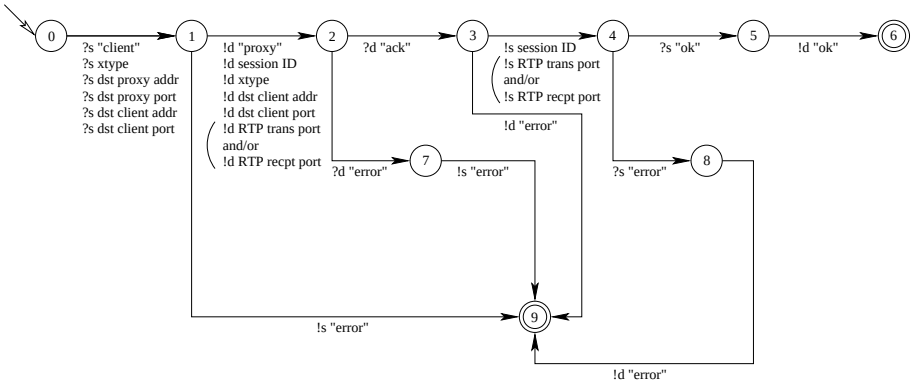

Figure 4.6: Source proxy finite state machine used during the initialization phase

Figure 4.7 describes what happens on a destination proxy when a communication initiation occurs. State number 0, is the state in which a proxy is when started. If state number 9 is reached, the communication attempt is discarded. State number 6 is the final state reached when the current communication establishment is a success. In the beginning, the destination proxy receives parameters from the source proxy. These are the *proxy* token which indicates that the entity on the other end of the connection is a source proxy, the `session ID`, the IP address and TCP port needed to reach a destination client and the RTP session port(s) (depending again on the requested communication type) for the communication between the two proxies. Once these data have been received (state number 1), the destination proxy attributes RTP session ports for its communication with the destination client. If a problem occurs, then it sends an *error* token to the source proxy and then goes to state number 9. If no problem occurs, then the destination proxy sends the communication parameters to the destination client. These are the `session ID` received from the source proxy, the requested communication type and the determined RTP session ports which will be used for the communication between the destination proxy and the destination client. At this point (state number 2), the destination proxy waits for the answer of the destination client. If an *error* token arrives (state number 7), then the same token is sent to the source proxy and the destination proxy goes

to state number 9. If an *ack* token arrives from the destination client (state number 3), then it is forwarded to the source proxy (state number 4). If an error occurs when the destination proxy is in state number 3, then an *error* token is sent to the destination client. This should never happen since the destination proxy doesn't perform anything critical in this state. When state number 4 is reached and an *ok* token arrives from the source proxy, it is forwarded to the destination client. Finally, the destination proxy can start its RTP session manager according to what has been configured in the previous steps.

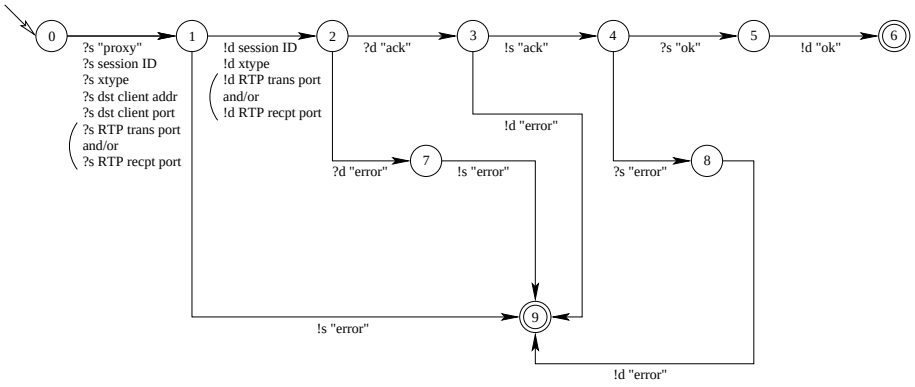

Figure 4.7: Destination proxy finite state machine used during the initialization phase

Figure 4.8 presents the initiation phase from the destination client point of view. A destination client is started with a single argument passed on the command line. This argument is the TCP port on which the destination client listens for connections made by a destination proxy. When started, a destination client is in state number 0 and waits for a connection from destination proxy. After a successful connection (state number 1), the connected destination proxy sends the communication parameters to the destination client. These are the `session ID`, the requested communication type and the RTP session ports which will be used for RTP exchange between the destination proxy and the destination client. Next, the destination client tries to start its configured RTP managers. If it fails to do it, it sends an *error* token to the destination proxy and is halted (state number 4). But, in case of success, it sends an *ack* token to the destination proxy and waits for the confirmation of the communication establishment success (state number 2). This confirmation is obtained by the reception of an *ok* token (state number 3).

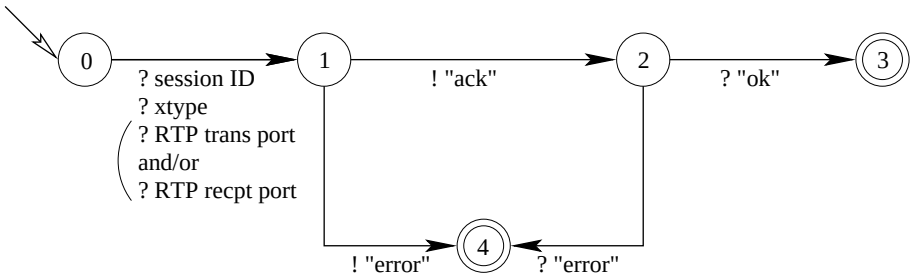

Figure 4.8: Destination client finite state machine used during the initialization phase

Figure 4.9 gives a global summary overview of the exchanges performed during the initialization phase. It shows that this phase is mainly composed of three sub-phases. The first one propagates the communication parameters on all the prototype components. The communication parameters transfered during this sub-phase are different for each step. First, the information transfered from the source client to the source proxy are:

- The IP address of the destination proxy and the TCP port on which it can be reached;
- The IP address of the destination client and the TCP port on which it can be reached;
- The requested communication type.

Next, the information transfered from the source proxy to the destination proxy are:

- The session ID;
- The IP address of the destination client and the TCP port on which it can be reached;
- The requested communication type;
- The RTP session port(s) for uni or bi-directional RTP communication between them.

Next, the information transfered from the destination proxy to the destination client are:

- The session ID;

- The requested communication type;
- The RTP session port(s) for uni or bi-directional RTP communication between them.

Finally, the information transfered from the source proxy to the source client are:

- The session ID;
- The RTP session port(s) for uni or bi-directional RTP communication between them.

The second sub-phase tells all the components that they can start their RTP session managers. And the last one indicates them that all the RTP session managers needed for the current communication have been successfully started. If any error occurs during any of these sub-phases, the global phase is then interrupted.

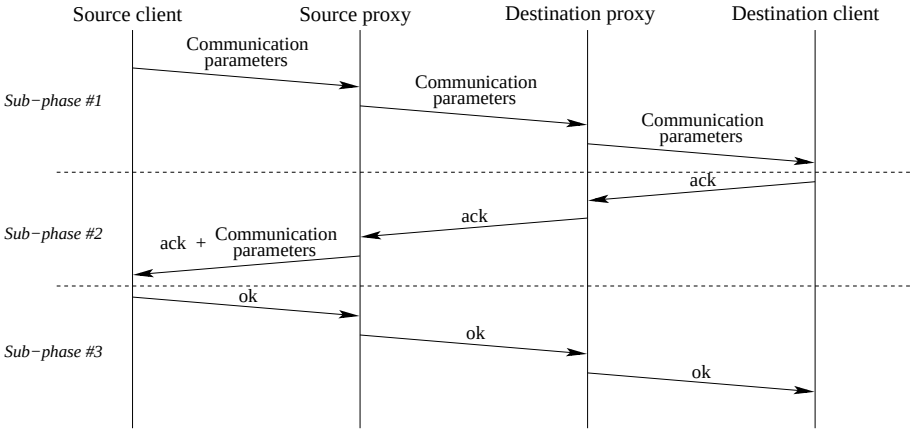

Figure 4.9: Communication parameters exchange summary

4.3.3 Client internals

This section presents the internal architecture of the client program evoked above. The source code files of this program are reproduced in the annex chapter in section A.1.1.

This program is articulated around three main classes:

- `ClientMain`;

- `ClientRTPTransmitter`;
- `ClientRTPReceiver`.

`ClientMain` is the class containing the entry point of the program. Its goal is to perform the parameters negotiation described above and to start the appropriate RTP components accordingly. These components are the two other main classes of the program: the `ClientRTPTransmitter` and the `ClientRTPReceiver`.

The `ClientRTPTransmitter` class is used to send the RTP data to the adjacent proxy. It is simply accomplished by connecting the data input of a `Processor` to the data output of an available capture device. A `SendStream` is then created with the data output of this `Processor` by a call to the `createSendStream()` method of a `RTPMnager` initialized with the considered RTP session parameters.

The `ClientRTPReceiver` class is used to receive the RTP flow sent by the adjacent proxy and to present the content of this flow to the user. This is done by connecting a `Processor` to the incoming RTP flow. The output of this `Processor` is then connected to a player able to present the received data.

Another class is used to initialize the different `RTPManagers`, the `ClientRTPSocketAdapter.java`. This class is the copy of a class used by the sample code provided by Sun Microsystems. It is reproduced in section A.1.1 and is available in its original form at `http://java.sun.com/products/java-media/jmf/2.1.1/solutions/RTPSocketAdapter.java`. It is also used in the proxy server and in the simulator.

4.3.4 Proxy server internals

The code of this program is reproduced in section A.1.2. It is composed of five main classes:

- `ProxyServer`;
- `ProxySession`;
- `ProxyRTPSessionsManager`;
- `H263VideoSigner`;
- `H263VideoVerifier`.

As we've already said, the proxy servers are able to manage multiple communications. The mechanisms used to do this are implemented in the `ProxyServer` class. This class holds the entry point of the program. When started, it waits for connection coming from proxies or clients. It creates a `ProxySession` object for each new connection. The maximum number of `ProxySession` existing at a given time can be specified on the command line when the considered proxy is launched.

Each `ProxySession` object created in this way performs the parameters negotiation described in section 4.3.2. When all the parameters are known, it creates a `ProxyRTPSessionsManager` which is then used to manage the requested RTP sessions. For each RTP session, a `Processor` is created. Its data input is connected to the `ReceiveStream` of the RTP session from which the flow is received. And its data output is connected to the `SendStream` of the RTP session on which it sends data.

When the considered flow originates from a client, the `Processor` is configured to make use of the `H263VideoSigner`. But if the flow originates from a proxy, the `Processor` is configured to use the `H263VideoVerifier` plugin.

The `H263VideoSigner` plugin introduces authentication information into the flow. The `H263VideoVerifier` removes them and checks for the validity of the received data. Those two plugins implement the tree-chaining authentication scheme. All the data entering in the `H263VideoVerifier` plugin are recorded as files. Thus, as their authenticity can be proved by using the recording checker program described in section 4.3.6, they can be, for instance, incorporated in electronic contracts.

We also implemented some of the other schemes: the basic scheme, the simple chaining scheme, EMSS and the star-chaining schemes. They were only implemented for simulation purposes[10]. However, as we designed them using the JMF plugin architecture, they can be easily used with the prototype. But, it must noted that we discovered security issues in our implementation of EMSS and of the simple chaining scheme. These are exposed in section 4.4. Their implementation can be found in section A.2.3 and A.2.4.

4.3.5 Key pair generator

This program is a simple utility which can be used to generate a DSA key pair. It is invoked on the command line with two arguments which are the names of the files intended to contain the generated private and public keys. Its code is reproduced in section A.1.3.

[10] See sections 5.3.3 and 5.3.4.

4.3.6 Recording checker

The code of this program is reproduced in section A.1.4. It takes for argument the name of a file containing the recording of a past communication. In order to authenticate the data contained in this file, it needs the public key corresponding to the private key which was used to sign the data.

This program is not based on JMF. Indeed, the data stored by the proxies in the recording files are simply `byte` array objects. Then, the recording checker program only consists in a loop reading one `byte[]` from the file and passing it through a rewriting of the `H263VideoVerifier` plugin at each iteration.

4.4 Security considerations

We already have identified two security issues in our authentication schemes implementation.

Firstly, let's consider a flow authenticated by either the simple chaining scheme or EMSS. There exists two possible reactions the system can have when it notes that a signed datagram is missing. It can stop the current transmission, considering that the chain (or multi-chain) this signed datagram authenticate is potentially composed of forged datagrams introduced by an attacker. Or it can simply let the communication go on. We choose to take the second option for our implementation. Indeed, we consider that the introduction of a small amount of datagrams forged by an attacker inside a video stream is clearly not enough to mislead a user. But it can constitute a serious threat for certain kinds of applications.

Secondly, the EMSS implementation makes intensive use of queues to store the payload of incoming datagrams. These payloads are stored until enough information have been gathered allowing their authentication to be accomplished. It appears that sometimes, the datagrams containing the information allowing to authenticate a waiting payload are lost. This implies that this payload will stay indefinitely into the queue. Now, let's suppose that an attacker starts sending forged datagrams that are valid for the application, but impossible to authenticate. Their payload will simply be stored into the queue which will grow until there is no memory left. This constitutes a denial of service attack. The whole system won't respond anymore. This is only an implementation design problem. It can be avoided by introducing more information into the datagrams and accepting only the datagrams respecting some criteria.

We propose a mechanism similar to the sliding window principle used

in TCP to solve this issue. Let's consider the embedding of an encrypted sequence number into the sent datagrams. Each receiver should be able to decrypt it. This can be done by the use of a symmetric cryptosystem for which the secret key should be securely distributed among the different communication participants. The Needham-Schroeder protocol can be used to accomplish it.

Let M be the maximum number of datagrams that we can accept at a time, this is the size of the sliding window we consider. This value is chosen in function of the number of datagrams needed to authenticate the content of a given datagram. Let P be the sequence number of the first datagram the considered receiver is ready to accept. Figure 4.10 illustrates this situation. The horizontal curve represents the range of the sliding window. For each incoming datagrams with sequence number i, three actions can be taken:

1. If $i < P$ the datagram is too late and is then dropped;
2. If $P < i < P + M - 1$ then the datagram is accepted and inserted into the queue;
3. If $P + M - 1 < i$ then the datagram is accepted and inserted into the queue. The sliding window is then moved to the right in order for sequence number i to be the greater acceptable sequence number of the sliding window. The data that are not anymore into the range of the sliding window are then removed from the queue.

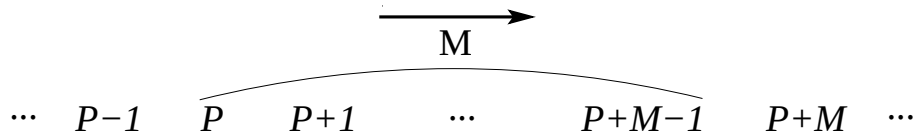

Figure 4.10: Sliding window principle

Despite the presence of this mechanism, there is still a problem. Indeed, an attacker can locate the encrypted portion of the datagrams and replace it by, for instance, random data. This would have for consequence that some datagrams forged by this attacker would still be inserted into the queue. To solve this issue, we can add a constant number chosen by convention along with the sequence number into the encrypted portion of the datagrams. In this way, before trying to guess what to do with an incoming datagram, the receiver would first have to check for the presence of the magic number. With this technique, the probability to see an undesirable datagram entering the queue is not null, but is really low.

A third security issue exists. It concerns the way the parameters are exchanged between the different components involved in a communication. Indeed, all the exchanges performed during the parameters negotiation phase are done without the use of any encryption mechanism. It means that anyone could easily retrieve the exchanged parameters and even try to accomplish a man in the middle attack against the two proxies involved in a communication. This can be solved by simply encrypting all the exchanges performed during the negotiation phase.

4.5 Conclusion

In this chapter we have presented the application we developed. First, we exposed its global structure and described its utilization.

Secondly, we detailed the parts of the JMF API which constitute the core of the whole prototype. We have presented the utility of the different JMF managers and the functioning of the JMF players and processors which are intensively used in our implementation.

Next, we described the negotiation phase performed for the establishment of the communication parameters. The goal of this phase, is to set up the whole system before the beginning of the communication. We described the finite-states machines used during this phase for the setup of the different components involved in a communication.

The internal functioning of the different components (programs) is then described by presenting the way we used JMF.

Finally, we have highlighted some security issues found in our implementation of some authentication schemes. These issues are mainly relative to this particular implementation. The first one concerns the way the simple chaining and EMSS schemes consider losses of signed packets. The second issue is the consequence of the lack of mechanism for the acceptance of incoming datagram into a queue. And the third is the consequence of the lack of encryption of the data exchanged during the paramaters negotiation phase. We proposed solutions to solve these problems.

Chapter 5

Implementation evaluation

5.1 Introduction

In this chapter, we give an evaluation of the performances of the Java implementation we proposed in the previous chapter. This evaluation is based on simulations of real situations. We start with the presentation of the simulation context. Next, we describe the simulation process and give the results. Finally, we discuss some security concerns with this implementation.

5.2 Simulation context

The simulations were performed with two computers connected to a fast Ethernet local area network. Figure 5.1 represents this physical context. Station A is an Intel Pentium III 500 MHz running a Redhat Linux 7.3 distribution (kernel 2.4.18). Station B is an AMD Duron 900 MHz running a Redhat Linux 7.1 distribution (kernel 2.4.2). Both machines are equipped with 256 MB of central memory. A video capture device[1] is connected to station A.

The Java Development Kit (JDK) used is version 1.4[2]. And the JMF implementation version is 2.2.1[3]. Both packages are available from Sun Microsystems.

The average network load measured, apart from the simulation traffic, was about 15 Kbytes/sec. This traffic was mainly composed of HTTP transactions and of traffic induced by audio streaming applications. It implies that

[1] It consists in a Brooktree Corporation Bt878 PCI card connected to a video recorder.
[2] See `http://java.sun.com/`
[3] See `http://java.sun.com/products/java-media/jmf/index.html`

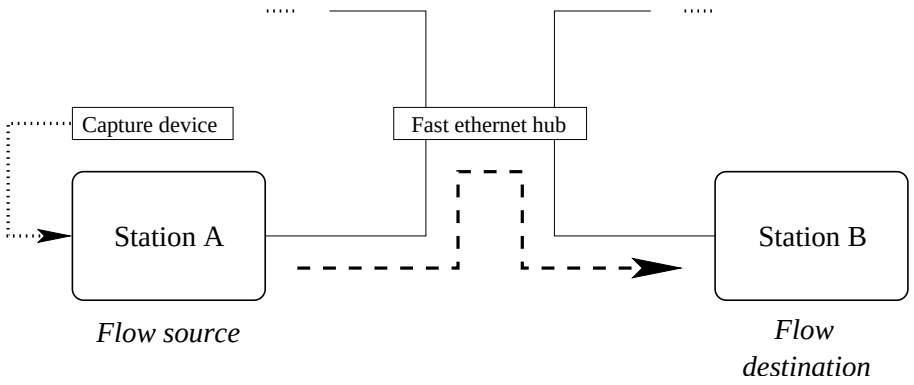

Figure 5.1: Simulation network

some of the datagrams transmitted by the application used for simulations were delayed because of frame collisions on the fast Ethernet link.

The CPU load observed when nothing was running on the machines, except the system daemons, was less than 2%.

5.3 Simulations

5.3.1 Hash value computation speed

In this section, we measure the execution speed of the methods used for fingerprint computation. This is done for both the MD5 and SHA1 JDK 1.4 implementations. The simulations performed in this section were accomplished on station A.

Since Linux is a multi-tasking operating system, delay evaluation cannot be done by simply taking absolute timestamps before and after the instruction (or sequence of instructions) we consider. By absolute timestamps, we mean timestamps relative to the system clock. Indeed, the task switching process performed by the operating system can disturb the measurement. The code example presented in figure 5.2 illustrates this.

In this example, we try to evaluate the time spent in the processing of the instruction on line number 5. We then take timestamps before (line number 4) and after (line number 6) this instruction. The problem occurs if, for instance, the operating system decides to perform task-switching during the period starting just after the execution of the instruction on line number 4 and ending just before the execution of the instruction on line number 6. Indeed, if the CPU resource is given to another process during this period, the measurement will be biased.

```
...

01 byte[] data = getData();
02 digester.update(data);
03
04 long ts1 = System.currentTimeMillis();
05 byte[] hash = digester.digest();
06 long ts2 = System.currentTimeMillis();

...
```

Figure 5.2: Task switching process can disturb delay measurement

To avoid this problem, we choose to use a code profiling tool called JMP[4]. This code profiler can be used, for example, to monitor objects creation and destruction and to measure the time actually spent into methods execution. JMP reports these time durations in microseconds, whereas Java is only able to work at the millisecond.

In order to perform this evaluation, we recorded 5000 JMF buffers containing real video data coming from the capture device. We observed that the average size of each data buffer was about 763 bytes. The hash values of all these records were computed with a simple program[5] composed of a loop which was computing a single hash value at each iteration. This program was then executed through JMP. Table 5.1 shows the results for the 5000 records. Column *Method* corresponds to the time spent in the method body and column *Sub-methods* corresponds to the time spent in the methods called by the considered one.

	Method (*sec*)	Sub-methods (*sec*)	Total (*sec*)
MD5	0.065231	6.193743	6.258974
SHA1	0.064311	2.822121	2.886432

Table 5.1: Actual time spent to compute 5000 hash values

The mean time for the computation of the MD5 hash value of a single record is 1252 microseconds. For the SHA1 it is only of 577 microseconds. These results can sound weird since the MD5 algorithm is known to be faster than SHA1. For example, the execution of the Unix command md5sum on a 35 Mbytes file takes about 2,44 seconds to complete on station A. The

[4]Available from `http://www.khelekore.org/jmp/index.html`

[5]See section A.2.1

execution of the sha1sum command on the same file with the same machine takes 4.3 seconds. Those two commands are implemented in C. Their source code is available from `http://www.gnu.org/software/textutils/textutils.html`.

It appears that the relative performances of those two algorithms depend on the size of the treated data buffer. Figure 5.3 shows the output of an application[6] that is built to benchmark MD5 and SHA1. These benchmarks show that MD5 is slower than SHA1 for small amounts of data, but that MD5 is clearly the fastest for data buffers larger than 32 Kbytes. The MD5 and SHA1 implementation used for these benchmarks are not the same as the ones coming from the JDK 1.4. These are custom implementations based on [18] for MD5 and on a public-domain C implementation avalaible from `http://www.mirrors.wiretapped.net/security/cryptography/hashes/sha1/sha1.c` for SHA1.

5.3.2 Signature computation and verification speed

These simulations were performed in the same conditions as those for the speed evaluation of the hashing algorithms implementation. They were done on the same machine and on the same 5000 JMF video data buffers. JMP was also used to evaluate method execution time for the same reasons as evoked above.

The program used for simulation also consists in a simple loop performing a signature and its verification on a single data buffer at each iteration. Its code is reproduced in section A.2.2.

The tests were realized on the RSA and DSA signature schemes supported by the Java standard implementation. Both of them are using SHA1 to compute the fingerprint of the data buffer they have to sign.

Table 5.2 shows the results. The *Sub-methods* column gives the time spent in the methods called by the signature computation or verification primitives. Some of the called methods are in this case the methods used to compute the fingerprint of the data buffer which must be signed or verified. Indeed, the signature computation and verification primitives are not directly applied on the data buffer but on its fingerprint. But, the majority of the *Sub-methods* are methods allowing the manipulation of big integers which are intensively used in public-key cryptography.

For the signing operation, DSA is almost twice as fast as RSA. But RSA is clearly better at verification. Data buffer length influence over the signing or verification operation is not significant. Indeed, it only influences the

[6] It can be found at `http://www.mcmanis.com/~cmcmanis/java/`

```
Benchmarking MD5
Bytes   Time (msec)   Rate (bytes/sec)
16384     90          182044.4444444444
32768      6         5461333.3333333333
49152      9         5461333.3333333333
65536     12         5461333.3333333333
81920     15         5461333.3333333333
98304     18         5461333.3333333333
114688    31         3699612.9032258065
131072    24         5461333.3333333333
147456    28         5266285.714285715
Done.

Benchmarking SHA1
Bytes   Time (msec)   Rate (bytes/sec)
16384     14         1170285.7142857143
32768     12         2730666.6666666665
49152     19         2586947.3684210526
65536     25         2621440.0
81920     53         1545660.3773584906
98304     38         2586947.3684210526
114688    45         2548622.2222222222
131072    50         2621440.0
147456    58         2542344.827586207
Done.
```

Figure 5.3: Benchmarks of MD5 and SHA1

		Method (sec)	Sub-methods (sec)	Total (sec)	Mean time (sec)
Signature	RSA	0.110208	8905.055751	8905.165959	1.781033
	DSA	0.117904	4923.178773	4923.296677	0.984659
Verification	RSA	0.099185	293.426101	293.525286	0.058705
	DSA	0.127448	9641.914938	9642.042386	1.928408

Table 5.2: Actual time spent to compute and verify 5000 signatures. Last column gives the mean time for the execution of the considered operation on a single data buffer.

fingerprint computation which is a very small part of the time spent for the whole operation (signature computation or verification).

5.3.3 Authentication time overhead

For these simulations, another set of program was written. It simply consists in two programs: a client and a server. Their code is reproduced in section A.2.3 and A.2.4. It constitutes a simplification of the prototype. The client is only able to send a RTP video data stream with any authentication scheme applied on it, and the server is only able to verify and present it to the user. The two stations available on the test network were then used for these simulations. Station A was the sender of the flows and station B was the receiver.

The delay introduced in a flow by an authentication scheme can be separated in two parts. These are the delay introduced on the sender's side and the delay introduced on the receiver's side. In order to measure these delays, we recorded the timestamps provided by calls to the system clock placed at the entry point and just before the end of the JMF plugins implementing the authentication schemes. The code profiler was not used mainly for two reasons. Firstly, this time overhead evaluation is done in order to reflect the delay that a user may experience in real situation and secondly, since JMF is a quite complex multi-threaded Java extension, running this simulation through JMP would have taken too much time. The tests were done over 5000 JMF buffers containing real video data.

Table 5.3 presents the results of these simulations. It appears that timestamping at the millisecond wasn't fine enough to estimate the delays for most of the schemes. We can only see that the basic scheme, the star and the tree-chaining scheme take way much time, compared to the other, to accomplish the verification of the authentication information. This can be improved by

the implementation of the caching technique exposed in section 3.6 for the tree-chaining scheme.

Authentication scheme		Average time overhead
Sender	Basic	44 ms
	Simple	< 1 ms
	Star	< 1 ms
	Tree	< 1 ms
	EMSS	< 1 ms
Receiver	Basic	37 ms
	Simple	< 1 ms
	Star	37.6 ms
	Tree	50.7 ms
	EMSS	< 1 ms

Table 5.3: Average time overhead introduced by the different authentication schemes (in milliseconds).

5.3.4 Authentication space overhead

In this section we evaluate, in terms of packet size, the mean overhead introduced, in a flow, by the different authentication schemes implementations. The content of the packets of this flow was composed of video data coming from the capture device.

The tests were done by using the same simulator as the one used for time overhead evaluation. But, instead of recording timestamps, the simulator was programmed to record JMF buffer sizes. 5000 buffers of real video were treated like this for each authentication schemes.

Table 5.4 shows the results. The *Bare* line gives the measurements obtained from the simulation performed without authentication scheme. The average size overhead column represents the difference between the *Bare* average packet size and the average packet size for the considered scheme.

Star and tree-chaining are, as we could expect, the schemes which introduce the largest size overhead. Basic, simple chaining and EMSS schemes size overheads are of a rather similar amplitude.

Authentication scheme	Average packet size	Average size overhead
Bare	763 bytes	0 bytes
Basic	816 bytes	53 bytes
Simple	795 bytes	32 bytes
Star	937 bytes	174 bytes
Tree	868 bytes	105 bytes
EMSS	812 bytes	49 bytes

Table 5.4: Average size overhead introduced by the different authentication schemes

5.4 Conclusion

In this chapter, we have performed various simulations in order to evaluate the performance of the authentication schemes that we have implemented.

We have run a first set of tests to provide an overview of the performance of the standard Java implementation of hashing and signing algorithms. These tests have shown that the MD5 implementation is slower than the one of SHA1 for data block size equivalent to the mean size of the data buffers actually processed by our prototype. The benchmarking of RSA and DSA Java standard implementations showed that RSA is slower than DSA for the signing operation, but that it is really faster than DSA for signature verification.

The second part of this chapter has presented the evaluation of our implementation of the authentication schemes. The mean delay and size overheads they introduce in a flow have been estimated. We were not able to measure, with enough precision, the mean time overhead introduced on the sender's side. Indeed it appears that this time overhead is less than one millisecond for each scheme, except for the basic one. On the receiver's side, the tree-chaining scheme introduces the largest mean time overhead. It is directly followed by star-chaining and basic scheme. The time granularity was not fine enough to estimate the mean time spent for authentication for the simple chaining and EMSS schemes.

Star-chaining is the scheme that introduces the largest mean size overhead. It is 65% larger than the size overhead introduced by the tree-chaining scheme. They are followed, in the order, by the basic, EMSS and simple chaining schemes.

Chapter 6

Conclusion and further work

Throughout this paper, we have presented our work on the authentication and non-repudiation problems for real-time flows at the application level.

Public-key cryptography can be used to address the problem of data authentication. It provides a mean to bind some data to the identity of a sending entity. Furthermore, the integrity of these data is also guaranteed.

We have highlighted the different key properties of a good solution. Firstly, the delay introduced by the different operations used to set up the authentication mechanism should be as small as possible. This is required in order to limit the impact on the applications. Secondly, the datagram size overhead introduced by the authentication scheme should be as small as possible in order to limit the utilization of the bandwidth. Thirdly, the authentication information inter-dependencies should be as limited as possible, and this in order to speed up the authentication process on both sides of the communication. Fourthly, the signature amortizing ratio of the considered authentication scheme should be balanced. It measures the ratio between a certain number of signatures and the number of datagrams they authenticate. It shouldn't be too small since it would mean that loss resistance would be weak, but on the other hand, it should not be too large since it would mean that there is a waste of signature. Finally, the loss resistance of the authentication scheme should be acceptable. This means that it should be as resistant as the application that makes use of it.

Real-time flow Authentication methods

Different schemes for the authentication of real-time flows using public-key cryptography are proposed in the literature. They all constitute optimizations of the naïve solution consisting in the signature of each datagram of

the considered flow. This solution is, indeed, fully resistant to losses, but, because of the intensive use of public-key cryptography, imposes significant CPU load on both the sender and the receiver of the flow.

The optimizations of the basic solution can be classified in two classes. The first approach consist in the introduction of hash value links between the datagrams. This leads to two different solutions: the simple chaining which consists in building simple datagram chains in a flow and the multi-chaining which consists in building several interlaced datagram chains in a flow in order to maximize the authentication probability of each datagram.

The second approach consists in splitting the flow in small blocks of a few datagrams and building a logical structure on top of it which allows to determine what will be the composition of the outgoing datagrams. This technique comes in two flavor: the star-chaining and the tree-chaining which is an optimization of the previous one.

Implementation and evaluation

We have implemented a prototype application allowing to perform authenticable H263 video communications. Our prototype is composed of two main programs: a client and a proxy server. They work in pair, a client together with a proxy server. Each pair is intended to be placed on a trusted network. Whenever a client (the source client) decides to contact another client (the destination client), he sends a request to the proxy server (the source proxy) he can reach through the trusted network. This source proxy then tries to contact the destination client by the intermediary of the specified proxy server (the destination proxy). Information are then exchanged in order to set up the video communication. Once all the components have been configured, the communication can begin.

The communications are recorded as files on the destination proxies along with the information allowing their authentication. We implemented a recording checker. It provides the ability to check the authenticity of the files containing the recorded communications.

The core of the prototype is based on the JMF API. This provides the plugin architecture on which we implemented different authentication schemes. As these plugins make use of public-key cryptography, they use key-pairs. We wrote a small utility program allowing to generate such key-pairs.

All the authentication schemes were implemented except the graph-based one.

Two characteristics of the different authentication schemes we have implemented were tested. These are the average time and size overhead they

introduce in the treated flows. We tested them by performing experimentation on 5000 real video data buffers.

The average time overhead evaluation is two-fold. It consists in the measurement of the time spent into the phase during which the authentication information are determined and introduced in the flow (on the sender's side) and the measurement of the time spent to verify and remove these authentication information (on the receiver's side). The time measurements were performed by taking a time reference at the beginning and at the end of the period we wanted to estimate. From the measurements performed for the first phase, we can conclude that the worst scheme implementation is the basic one. It is a normal observation, since this scheme compute a signature for each datagram. The evaluation of the other schemes for this phase was not concluding. Indeed, the time granularity wasn't fine enough to allow accurate estimations to be performed on the sender's side. All that we known is that the average time spent for the authentication phase on the sender's side for all the other schemes is smaller than one millisecond.

On the other hand, on the receiver's side the results were more interesting. The most greedy scheme is the tree-chaining. It is followed by the star-chaining and the basic schemes. This observation is quite normal since star and tree-chaining perform a lot of operations for the treatment of each datagrams where the basic scheme compute a signature for each datagram. We were not able to decide between the EMSS and the simple chaining schemes for the same reason as evoked above about time granularity.

The average size overhead evaluation was performed by measuring the mean buffer size increasing of 5000 real video data buffers for each authentication schemes compared to the situation where non authentication scheme is involved. This results show that the largest average size overhead is introduced by the star-chaining scheme. It is followed by the tree-chaining which is designed to limit the size overhead compared to star-chaining. Simple chaining introduces the smaller average size overhead. It is followed by the EMSS and the basic schemes.

We also tested the JDK 1.4 hashing and signing primitives implementation. This was done by performing the tests on actual H263 video data buffers by using the JMP code profiler. This tool has been used to observe the time spent in the execution of the considered methods.

We tested the implementation of MD5 and SHA1. It appears that the MD5 implementation is slower than the SHA1 implementation. Our analysis showed that this is the case because our tests were led on small, but realistic, data buffers. Indeed, we showed that MD5 is slower than SHA1 for data buffer sizes smaller than 32 KBytes.

The implementation of the RSA and DSA signature computation and

verification primitives was tested. It resorts that DSA is the fastest for signature computation, but not for signature verification.

Further work

The way the different authentication schemes are used in our prototype can be improved. They don't take advantage of all the information provided by the RTP protocol. For instance, we could imagine a mechanism allowing to adapt some parameters of the considered authentication scheme in function of the information reported by RTP on the quality of transmission. For example, in the case of the EMSS scheme, we could increase the signature frequency when the loss rate raises. This would have for consequence to improve the probability of a successful authentication for each incoming datagram.

The work we've done could also be extended by developing a more useful and usable utility allowing to deal with real situations needs, or by extending an existing functionnal video conferencing application such as Gnomemeeting. Gnomemeeting[1] is an audio/video conferencing application based on the H323 protocol[2]. It is free software written in C++. H323 is a protocol allowing to transport audio and video informations over a network. As Gnomemeeting doesn't provide any flow authentication capability, an interesting work would be to adapt to it some of the authentication schemes we have implemented.

[1] `http://www.gnomemeeting.org/`
[2] `http://www.openh323.org/`

Part III

Appendix

Appendix A

Source code

A.1 Prototype

A.1.1 Client

Prototype/Client/ClientMain.java

```
1  package Client;
2
3  import java.lang.*;
4  import java.io.*;
5  import java.net.*;
6  import java.util.*;
7
8  public class ClientMain {
9
10     public static ClientRTPTransmitter transmitter = null;
11     public static ClientRTPReceiver receiver = null;
12
13     public static void main(String args[]) {
14
15         // Socket used to communicate with the adjacent proxy
16         Socket soc = null;
17
18         // Object stream used to exchange Objects with the adjacent proxy
19         InputStream is = null;
20         OutputStream os = null;
21         ObjectInput ois = null;
22         ObjectOutput oos = null;
23
24         // The port on which the RTP data are sent to the adjacent proxy
25         // (value 0 means not set)
26         int RTPTransmissionPort = 0;
27
28         // The port on which the RTP data are received from the adjacent proxy
29         // (value 0 means not set)
30         int RTPReceptionPort = 0;
31
32         // IP address of the adjacent proxy
33         String proxyAddress = null;
34
35         // The communication type associated with this client
36         // can be either rx, tx or rxtx
37         String xType = null;
38
39         int sessionID = 0;  // The current session identifier
40                             // (value 0 means not set)
41
42         if(args.length == 1) {
43             // Destination client
44             // args[0] == listening port
45 // The state number evoked in this block refer to figure 4.8
```

```
46  // State number 0
47              info("This client is a destination client");
48              info("Listening port: " + args[0]);
49
50              // Wait for connection of a destination proxy
51              try {
52                  ServerSocket server = new ServerSocket(Integer.parseInt(args[0]));
53                  info("Waiting for destination proxy connection ...");
54                  soc = server.accept();
55                  proxyAddress = soc.getInetAddress().getHostAddress();
56              }
57              catch(IOException e) {
58                  error("I/O exception");
59              }
60
61              info("Destination proxy connected");
62
63              Vector dataVect = null;
64
65              // Receive the session ID, the exchange type
66              // and the RTP parameters from the connected destination proxy
67              try {
68                  is = soc.getInputStream();
69                  ois = new ObjectInputStream(is);
70                  dataVect = (Vector) ois.readObject();
71                  info("RTP parameters received from proxy");
72              }
73              catch (ClassNotFoundException e) {
74                  error("Class not found exception");
75              }
76              catch (IOException e) {
77                  error("I/O exception");
78              }
79
80              String xTypeTmp = null;
81
82              switch(dataVect.size()) {
83
84              case 3:
85                  sessionID = ((Integer) dataVect.get(0)).intValue();
86
87                  xTypeTmp = (String) dataVect.get(1);
88                  if(xTypeTmp.equals("tx"))
89                      xType = "rx";
90                  else if(xTypeTmp.equals("rx"))
91                      xType = "tx";
92                  else
93                      xType = xTypeTmp;
94
95                  if(xType.equals("tx"))
96                      RTPTransmissionPort = ((Integer) dataVect.get(2)).intValue();
97                  else
98                      RTPReceptionPort = ((Integer) dataVect.get(2)).intValue();
99                  break;
100
101             case 4:
102                 sessionID = ((Integer) dataVect.get(0)).intValue();
103
104                 xTypeTmp = (String) dataVect.get(1);
105                 if(xTypeTmp.equals("tx"))
106                     xType = "rx";
107                 else if(xTypeTmp.equals("rx"))
108                     xType = "tx";
109                 else
110                     xType = xTypeTmp;
111
112                 RTPTransmissionPort = ((Integer) dataVect.get(2)).intValue();
113                 RTPReceptionPort = ((Integer) dataVect.get(3)).intValue();
114                 break;
115
116             default:
117                 error("Bad proxy message");
118             }
119
120 // State number 1
121             info("RTP transmission port: " + RTPTransmissionPort);
122             info("RTP reception port: " + RTPReceptionPort);
123             info("Exchange type: " + xType);
124             info("Session ID: " + sessionID);
125
126             // Start the RTP transmission if required
127             String retValue = "OK";
128
```

```
129             if(xType.endsWith("tx")) {
130                 retValue = startTransmission(proxyAddress,RTPTransmissionPort);
131             }
132 
133             // Confirm, to the destination proxy, the success of RTP
134             // transmission starting
135             try {
136                 os = soc.getOutputStream();
137                 oos = new ObjectOutputStream(os);
138                 dataVect = new Vector();
139                 if(retValue.startsWith("OK"))
140                     dataVect.add(new String("ACK"));
141                 else
142                     dataVect.add(new String("Error"));
143                 oos.writeObject(dataVect);
144                 oos.flush();
145             }
146             catch (IOException e) {
147                 error("I/O exception");
148             }
149 
150             if(!retValue.startsWith("OK"))
151 
152 // State number 4
153                 error(retValue);
154 
155 // State number 2
156             // Receive the confirmation from the destination proxy
157             try {
158                 dataVect = (Vector) ois.readObject();
159             }
160             catch (ClassNotFoundException e) {
161                 error("Class not found exception");
162             }
163             catch (IOException e) {
164                 error("I/O exception");
165             }
166 
167             if(!((String) dataVect.get(0)).equals("OK"))
168 
169 // State number 4
170                 error("Error message received from destination proxy");
171 
172 // State number 3
173             // Start the RTP reception if required
174             if(xType.startsWith("rx")) {
175                 retValue = startReception(RTPReceptionPort);
176                 if(!retValue.startsWith("OK"))
177                     error(retValue);
178             }
179             else {
180                 // Catch a user interruption
181                 try {
182                     BufferedReader d =
183                         new BufferedReader(new InputStreamReader(System.in));
184                     String line;
185 
186                     while((line = d.readLine()) != null) {
187                     }
188                     if(line == null) {
189                         info("Ctrl-d key pressed");
190                         info("Stopping RTP transmitter");
191                         transmitter.stop();
192                     }
193                     d.close();
194                 }
195                 catch(IOException e) {
196                     error("I/O exception on standard input");
197                 }
198             }
199         }
200         else if(args.length == 7) {
201             // Source client
202             // args[0] == source proxy IP address
203             // args[1] == source proxy TCP port
204             // args[2] == destination proxy IP address
205             // args[3] == destination proxy TCP port
206             // args[4] == destination client IP address
207             // args[5] == destination client TCP port
208             // args[6] == rx | tx | rxtx
209 // The state number evoked in this block refer to figure 4.5
210 // State number 0
211             proxyAddress = args[0];
```

```
212             xType = args[6];
213
214             info("This client is a source client");
215
216             info("Source proxy IP address: " + args[0]);
217             info("Source proxy port: " + args[1]);
218             info("Destination proxy IP address: " + args[2]);
219             info("Destination proxy port: " + args[3]);
220             info("Destination client IP address: " + args[4]);
221             info("Destination client port: " + args[5]);
222
223             // Try to connect to source proxy
224             try {
225                 soc = new Socket(InetAddress.getByName(proxyAddress),
226                                  Integer.parseInt(args[1]));
227             }
228             catch (IOException e) {
229                 error("I/O error. Can't create socket to source proxy");
230             }
231
232             // Transmit the session parameters to source proxy
233             Vector dataVect = null;
234             try {
235                 os = soc.getOutputStream();
236                 oos = new ObjectOutputStream(os);
237
238                 dataVect = new Vector();
239                 dataVect.add(new String("Client"));
240                 dataVect.add(xType);
241                 dataVect.add(args[2]);
242                 dataVect.add(new Integer(Integer.parseInt(args[3])));
243                 dataVect.add(args[4]);
244                 dataVect.add(new Integer(Integer.parseInt(args[5])));
245                 oos.writeObject(dataVect);
246                 oos.flush();
247                 info("Session parameters sent to source proxy");
248             }
249             catch (IOException e) {
250                 error("I/O exception");
251             }
252
253 // State number 1
254             // Receive the session ID and the RTP parameters from proxy
255             try {
256                 is = soc.getInputStream();
257                 ois = new ObjectInputStream(is);
258
259                 dataVect = (Vector) ois.readObject();
260                 info("RTP parameters received from proxy");
261             }
262             catch (ClassNotFoundException e) {
263                 error("Class not found exception");
264             }
265             catch (IOException e) {
266                 error("I/O exception");
267             }
268
269             switch(dataVect.size()) {
270
271             case 1:
272                 error("Proxy error");
273
274 // State number 4
275                 break;
276
277             case 2:
278                 sessionID = ((Integer) dataVect.get(0)).intValue();
279                 if(xType.equals("tx"))
280                     RTPTransmissionPort = ((Integer) dataVect.get(1)).intValue();
281                 else
282                     RTPReceptionPort = ((Integer) dataVect.get(1)).intValue();
283                 break;
284
285             case 3:
286                 sessionID = ((Integer) dataVect.get(0)).intValue();
287                 RTPTransmissionPort = ((Integer) dataVect.get(1)).intValue();
288                 RTPReceptionPort = ((Integer) dataVect.get(2)).intValue();
289                 break;
290
291             default:
292                 error("Bad proxy message");
293             }
294
```

```
295             info("RTP transmission port: " + RTPTransmissionPort);
296             info("RTP reception port: " + RTPReceptionPort);
297             info("Exchange type: " + xType);
298             info("Session ID: " + sessionID);
299 
300 // State number 2
301             // Start the RTP transmission if required
302 
303             String retValue = "OK";
304 
305             if(xType.endsWith("tx")) {
306                 retValue = startTransmission(proxyAddress,RTPTransmissionPort);
307             }
308 
309             // Confirm, to the source proxy, the success of RTP transmission starting
310             try {
311                 dataVect = new Vector();
312                 if(retValue.startsWith("OK"))
313                     dataVect.add(retValue);
314                 else
315                     dataVect.add(new String("Error"));
316                 oos.writeObject(dataVect);
317                 oos.flush();
318             }
319             catch (IOException e) {
320                 error("I/O exception");
321             }
322 
323             if(!retValue.startsWith("OK"))
324 
325 // State number 4
326                 error(retValue);
327 
328 // State number 3
329             // Start RTP reception if required
330             if(xType.startsWith("rx")) {
331                 retValue = startReception(RTPReceptionPort);
332                 if(!retValue.startsWith("OK"))
333                     error(retValue);
334             }
335             else {
336                 // Catch a user interruption
337                 try {
338                     BufferedReader d =
339                         new BufferedReader(new InputStreamReader(System.in));
340                     String line;
341 
342                     while((line = d.readLine()) != null) {
343                     }
344                     if(line == null) {
345                         info("Ctrl-d key pressed");
346                         info("Stopping RTP transmitter");
347                         transmitter.stop();
348                     }
349                     d.close();
350                 }
351                 catch(IOException e) {
352                     error("I/O exception on standard input");
353                 }
354             }
355         }
356         else
357             // Print program usage to standard output
358             prUsage();
359 
360         try {
361             ois.close();
362             oos.close();
363             soc.close();
364         }
365         catch (IOException e) {
366             error("I/O exception");
367         }
368         System.exit(0);
369     }
370 
371     // Create a RTP transmitter and start the RTP transmission
372     static String startTransmission(String proxyAddress, int RTPTransmissionPort) {
373 
374         transmitter = new ClientRTPTransmitter(proxyAddress,RTPTransmissionPort);
375 
376         info("Starting RTP transmission ...");
377         String result = transmitter.start();
```

```
378
379         if (result != null) {
380             return(result);
381         }
382
383         return("OK");
384     }
385
386     // Create a RTP receiver, start it and wait for a user interruption
387     static String startReception(int RTPReceptionPort) {
388
389         try {
390             receiver =
391                 new ClientRTPReceiver(InetAddress.getLocalHost().getHostAddress(),
392                                       RTPReceptionPort);
393         }
394         catch(UnknownHostException e) {
395             return("Unknown host exception");
396         }
397         info("Starting RTP reception ...");
398         if(!receiver.initialize()) {
399             return("Failed to start RTP receiver");
400         }
401
402         try {
403             BufferedReader d = new BufferedReader(new InputStreamReader(System.in));
404             String line;
405
406             while(((line = d.readLine()) != null) && (!receiver.isDone())) {
407             }
408             if(line == null) {
409                 info("Ctrl-d key pressed");
410                 info("Stopping RTP receiver and transmitter");
411                 receiver.close();
412                 if(transmitter != null)
413                     transmitter.stop();
414             }
415             d.close();
416         }
417         catch(IOException e) {
418             error("I/O exception on standard input");
419         }
420         return("OK");
421     }
422
423     // Print program usage on standard output
424     static void prUsage() {
425         info("Usage:");
426         info("Source client mode");
427         info("java Client.ClientMain <srcProxyIp> <srcProxyPort> <dstProxyIp> " +
428              "<dstProxyPort> <dstClientIp> <dstClientPort> <xType>");
429         info("      <srcProxyIp>: source proxy IP address");
430         info("      <srcProxyPort>: source proxy TCP port");
431         info("      <dstProxyIp>: destination proxy IP address");
432         info("      <dstProxyPort>: destination proxy TCP port");
433         info("      <dstHostIp>: destination client IP address");
434         info("      <dstHostPort>: destination client TCP port");
435         info("      <xType>: exchange type (rx, tx or rxtx)");
436         info("Destination client mode");
437         info("java Client.ClientMain <listeningPort>");
438         info("      <listeningPort>: tcp port number listened for proxy connection");
439
440         System.exit(0);
441     }
442
443     static private void info(String msg) {
444         System.out.println("[INFO] Client: " + msg);
445     }
446
447     static private void error(String msg) {
448         System.err.println("[ERROR] Client: " + msg);
449         System.exit(1);
450     }
451 }
```

Prototype/Client/ClientRTPTransmitter.java

```
1  package Client;
2
3  import java.awt.*;
4  import java.io.*;
5  import java.util.*;
6  import java.net.InetAddress;
7  import javax.media.*;
8  import javax.media.protocol.*;
9  import javax.media.protocol.DataSource;
10 import javax.media.format.*;
11 import javax.media.control.*;
12 import javax.media.control.TrackControl;
13 import javax.media.control.QualityControl;
14 import javax.media.rtp.*;
15 import javax.media.rtp.rtcp.*;
16 import com.sun.media.rtp.*;
17
18
19 public class ClientRTPTransmitter {
20
21     // IP address and UDP port of the adjacent proxy
22     // used for the RTP transmission
23     private String ipDst;
24     private int portDst;
25
26     // JMF processor used to interconnect the capture device and the RTPManager
27     private Processor processor = null;
28     // JMF RTPManager used to transmit data over RTP
29     private RTPManager rtpMgr = null;
30     // JMF DataSource used to handle the data coming out from the processor
31     private DataSource dataOutput = null;
32
33     // Constructor
34     public ClientRTPTransmitter(String ipDst, int portDst) {
35         this.ipDst = ipDst;
36         this.portDst = portDst;
37     }
38
39     // Start the RTP transmission
40     // Returns null if transmission started successfully
41     // Otherwise it returns a string with the reason why it failed
42     public synchronized String start() {
43         String result;
44
45         // Create and configure the processor used to interconnect
46         // the capture device and the RTPManager
47         result = createProcessor();
48         if (result != null)
49             return result;
50
51         // Create and initialize the RTPManager used to send the RTP data
52         result = createTransmitter();
53         if (result != null) {
54             processor.close();
55             processor = null;
56             return result;
57         }
58
59         // Start the transmission
60         processor.start();
61
62         return null;
63     }
64
65     // Stop the existing transmission
66     public void stop() {
67         synchronized (this) {
68             if (processor != null) {
69                 processor.stop();
70                 processor.close();
71                 processor = null;
72                 rtpMgr.removeTargets("Session ended");
73                 rtpMgr.dispose();
74             }
75         }
76     }
77
78     // Create a processor with for input the data coming out of the
79     // capture device and program it to output H263/RTP
80     private String createProcessor() {
```

```
81
82          DataSource ds;
83
84          try {
85              ds = createDataSource(new VideoFormat("RGB",
86                                    new Dimension(352, 288),
87                                    Format.NOT_SPECIFIED,
88                                    null,
89                                    Format.NOT_SPECIFIED));
90          }
91          catch(Exception e) {
92              return "Couldn't create DataSource";
93          }
94
95          // Try to create a processor to handle the data source just created
96          try {
97              processor = javax.media.Manager.createProcessor(ds);
98          }
99          catch(NoProcessorException npe) {
100             return "Couldn't create processor";
101         }
102         catch(IOException ioe) {
103             return "I/O exception creating processor";
104         }
105
106         // Wait for it to configure
107         boolean result = waitForState(processor, Processor.Configured);
108         if (result == false)
109             return "Couldn't configure processor";
110
111         // Get the tracks from the processor
112         TrackControl [] tracks = processor.getTrackControls();
113         if (tracks == null || tracks.length < 1)
114             return "Couldn't find tracks in processor";
115
116         // Set the output content descriptor to RAW_RTP
117         // This will limit the supported formats reported from
118         // Track.getSupportedFormats to only valid RTP formats
119         ContentDescriptor cd = new ContentDescriptor(ContentDescriptor.RAW_RTP);
120         processor.setContentDescriptor(cd);
121
122         Format supported[];
123         Format chosen;
124         boolean atLeastOneTrack = false;
125
126         // Program the tracks
127         for (int i = 0; i < tracks.length; i++) {
128             Format format = tracks[i].getFormat();
129
130             // Insert the encoding plugins into the video track plugin chain
131             // to force H263 encoding
132             if(format instanceof VideoFormat) {
133                 try {
134                     info("Setting encoding plugins chain for track " + i + " ...");
135                     tracks[i].setCodecChain(
136                             new Codec[] {
137                                 new com.sun.media.codec.video.colorspace.JavaRGBToYUV(),
138                                 new com.ibm.media.codec.video.h263.NativeEncoder()
139                             });
140                 }
141                 catch (NotConfiguredError e) {
142                     error("Not Configured Error");
143                 }
144                 catch (UnsupportedPlugInException e) {
145                     error("Unsupported PlugIn Exception");
146                 }
147             }
148
149             // Set the right video dimensions for the video track
150             if (tracks[i].isEnabled()) {
151
152                 supported = tracks[i].getSupportedFormats();
153
154                 if (supported.length > 0) {
155                     if (supported[0] instanceof VideoFormat) {
156                         chosen = checkForVideoSizes(tracks[i].getFormat(),
157                                                     supported[0]);
158                     } else
159                         chosen = supported[0];
160                     tracks[i].setFormat(chosen);
161                     info("Track " + i + " is set to transmit as:");
162                     info("  " + chosen);
163                     atLeastOneTrack = true;
```

```
164                 } else
165                     tracks[i].setEnabled(false);
166             } else
167                 tracks[i].setEnabled(false);
168         }
169 
170         if (!atLeastOneTrack)
171             return "Couldn't set any of the tracks to a valid RTP format";
172 
173         // Realize the processor
174         result = waitForState(processor, Controller.Realized);
175         if (result == false)
176             return "Couldn't realize processor";
177 
178         // Get the output data source of the processor
179         dataOutput = processor.getDataOutput();
180 
181         return null;
182     }
183 
184     // Create a data source connected to a capture device able to provide
185     // the given format
186     DataSource createDataSource(Format format) throws Exception {
187         DataSource ds;
188         Vector devices;
189         CaptureDeviceInfo cdi;
190         MediaLocator ml;
191 
192         // Find devices for format
193         devices = CaptureDeviceManager.getDeviceList(format);
194         if (devices.size() < 1) {
195             error("No Devices for " + format);
196             throw new Exception();
197         }
198         // Pick the first device
199         cdi = (CaptureDeviceInfo) devices.elementAt(0);
200 
201         ml = cdi.getLocator();
202 
203         ds = Manager.createDataSource(ml);
204         ds.connect();
205         if(ds instanceof CaptureDevice)
206             setCaptureFormat((CaptureDevice) ds, format);
207         return ds;
208     }
209 
210     // Set the output format of the given capture device to the specified format
211     void setCaptureFormat(CaptureDevice cdev, Format format) {
212         FormatControl [] fcs = cdev.getFormatControls();
213         if (fcs.length < 1)
214             return;
215         FormatControl fc = fcs[0];
216         Format [] formats = fc.getSupportedFormats();
217 
218         for (int i = 0; i < formats.length; i++) {
219             if (formats[i].matches(format)) {
220                 format = formats[i].intersects(format);
221                 info("Setting format " + format);
222                 fc.setFormat(format);
223                 break;
224             }
225         }
226     }
227 
228     // Create a RTP session to transmit the output of the
229     // processor to the specified IP address ipDst and port number portDst
230     private String createTransmitter() {
231 
232         SendStream sendStream;
233 
234         try {
235             rtpMgr = RTPManager.newInstance();
236 
237             // Initialize the RTPManager with the RTPSocketAdapter
238             rtpMgr.initialize(
239                         new ClientRTPSocketAdapter(InetAddress.getByName(ipDst),
240                                                    portDst));
241 
242             info("Created RTP session: " + ipDst + " " + portDst);
243 
244             sendStream = rtpMgr.createSendStream(dataOutput,0);
245 
246             // Start the transmission
```

```
247             sendStream.start();
248         } catch (Exception  e) {
249             return e.getMessage();
250         }
251         return null;
252     }
253 
254     // Check if the video dimensions of original format matches one of the H263
255     // standard video formats. Returns the most adequate format in relation
256     // with the supported format
257     Format checkForVideoSizes(Format original, Format supported) {
258 
259         int width, height;
260         Dimension size = ((VideoFormat)original).getSize();
261         Format h263Fmt = new Format(VideoFormat.H263_RTP);
262 
263         if (supported.matches(h263Fmt)) {
264             // H263 only supports specific dimensions
265             if (size.width < 128) {
266                 width = 128;
267                 height = 96;
268             } else if (size.width < 176) {
269                 width = 176;
270                 height = 144;
271             } else {
272                 width = 352;
273                 height = 288;
274             }
275         } else
276             return supported;
277 
278         return (new VideoFormat(null,
279                                 new Dimension(width, height),
280                                 Format.NOT_SPECIFIED,
281                                 null,
282                                 Format.NOT_SPECIFIED)).intersects(supported);
283     }
284 
285     private void info(String msg) {
286         System.out.println("[INFO] Client RTP Transmitter: " + msg);
287     }
288 
289     private void error(String msg) {
290         System.err.println("[ERROR] Client RTP Transmitter: " + msg);
291     }
292 
293     // Convenience methods to handle processor's state changes
294     private Integer stateLock = new Integer(0);
295     private boolean failed = false;
296 
297     Integer getStateLock() {
298         return stateLock;
299     }
300 
301     void setFailed() {
302         failed = true;
303     }
304 
305     private synchronized boolean waitForState(Processor p, int state) {
306         p.addControllerListener(new StateListener());
307         failed = false;
308 
309         // Call the required method on the processor
310         if (state == Processor.Configured) {
311             p.configure();
312         } else if (state == Processor.Realized) {
313             p.realize();
314         }
315         while (p.getState() < state && !failed) {
316             synchronized (getStateLock()) {
317                 try {
318                     getStateLock().wait();
319                 } catch (InterruptedException ie) {
320                     return false;
321                 }
322             }
323         }
324 
325         if (failed)
326             return false;
327         else
328             return true;
329     }
```

```
330
331     // Convenience class to handle processor's state changes
332     class StateListener implements ControllerListener {
333
334         public void controllerUpdate(ControllerEvent ce) {
335
336             // If there is an error during configure or
337             // realize, the processor will be closed
338             if (ce instanceof ControllerClosedEvent)
339                 setFailed();
340
341             // All controller events send a notification
342             // to the waiting thread in waitForState method
343             if (ce instanceof ControllerEvent) {
344                 synchronized (getStateLock()) {
345                     getStateLock().notifyAll();
346                 }
347             }
348         }
349     }
350 }
```

Prototype/Client/ClientRTPReceiver.java

```
1  package Client;
2
3  import java.io.*;
4  import java.awt.*;
5  import java.net.*;
6  import java.awt.event.*;
7  import java.util.Vector;
8
9  import javax.media.*;
10 import javax.media.rtp.*;
11 import javax.media.rtp.event.*;
12 import javax.media.rtp.rtcp.*;
13 import javax.media.protocol.*;
14 import javax.media.protocol.DataSource;
15 import javax.media.format.AudioFormat;
16 import javax.media.format.VideoFormat;
17 import javax.media.Format;
18 import javax.media.format.FormatChangeEvent;
19 import javax.media.control.BufferControl;
20 import javax.media.control.TrackControl;
21 import javax.media.control.QualityControl;
22
23
24 public class ClientRTPReceiver implements ReceiveStreamListener, SessionListener,
25                                           ControllerListener
26 {
27     // IP address and UDP port of the adjacent proxy
28     // used for the RTP reception
29     private String ipSrc;
30     private int portSrc;
31
32     // JMF RTPManager used to transmit data over RTP
33     private RTPManager mgr = null;
34     // Vector used to keep references of the created JMF players
35     private Vector playerWindows = null;
36
37     // Convenience variable and object used to manage the RTP data waiting loop
38     private boolean dataReceived = false;
39     private Object dataSync = new Object();
40
41     // Constructor
42     public ClientRTPReceiver(String ipSrc, int portSrc) {
43         this.ipSrc = ipSrc;
44         this.portSrc = portSrc;
45     }
46
47     // Initialize a RTPManager and open the RTP session
48     protected boolean initialize() {
49         try {
50             playerWindows = new Vector();
51
52             info("Opening RTP session for: addr: " + ipSrc + " port: " + portSrc);
53
54             // Initialize the RTPManager with the RTPSocketAdapter
55             mgr = (RTPManager) RTPManager.newInstance();
56             mgr.addSessionListener(this);
57             mgr.addReceiveStreamListener(this);
58             mgr.initialize(new ClientRTPSocketAdapter(InetAddress.getByName(ipSrc),
59                                                       portSrc,1));
60
61             // Set the input buffer size
62             BufferControl bc =
63                 (BufferControl) mgr.getControl("javax.media.control.BufferControl");
64             if (bc != null)
65                 bc.setBufferLength(1000);
66
67         } catch (Exception e){
68             error("Cannot create the RTP session: " + e.getMessage());
69             close();
70             return false;
71         }
72
73         // Wait for data to arrive
74         long then = System.currentTimeMillis();
75         // Wait for a maximum of 300 secs
76         long waitingPeriod = 300000;
77
78         try{
79             synchronized (dataSync) {
80                 while (!dataReceived &&
```

```
81                      System.currentTimeMillis() - then < waitingPeriod) {
82                  if (!dataReceived)
83                      info("  - Waiting for RTP data to arrive...");
84                  dataSync.wait(1000);
85              }
86          }
87      } catch (Exception e) {}
88
89      if (!dataReceived) {
90          error("No RTP data were received.");
91          close();
92          return false;
93      }
94      return true;
95  }
96
97  // Check if all the players are closed
98  public boolean isDone() {
99      return playerWindows.size() == 0;
100 }
101
102 // Close the players and the session manager
103 protected void close() {
104
105     for(int i = 0; i < playerWindows.size(); i++) {
106         try {
107             ((PlayerWindow)playerWindows.elementAt(i)).close();
108         } catch (Exception e) {}
109     }
110
111     playerWindows.removeAllElements();
112
113     // Close the RTP session.
114     if (mgr != null) {
115         mgr.removeTargets("Closing RTP session");
116         mgr.dispose();
117         mgr = null;
118     }
119 }
120
121 // Returns the player window of a given player
122 PlayerWindow find(Player p) {
123     for (int i = 0; i < playerWindows.size(); i++) {
124         PlayerWindow pw = (PlayerWindow)playerWindows.elementAt(i);
125         if (pw.player == p)
126             return pw;
127     }
128     return null;
129 }
130
131 // Returns the player window of a given RTP receive stream
132 PlayerWindow find(ReceiveStream strm) {
133     for (int i = 0; i < playerWindows.size(); i++) {
134         PlayerWindow pw = (PlayerWindow)playerWindows.elementAt(i);
135         if (pw.stream == strm)
136             return pw;
137     }
138     return null;
139 }
140
141 // RTP session listener
142 public synchronized void update(SessionEvent evt) {
143     if (evt instanceof NewParticipantEvent) {
144         Participant p = ((NewParticipantEvent)evt).getParticipant();
145         info("A new participant had just joined: " + p.getCNAME());
146     }
147 }
148
149 // RTP receive stream listener
150 public synchronized void update(ReceiveStreamEvent evt) {
151
152     RTPManager mgr = (RTPManager)evt.getSource();
153     Participant participant = evt.getParticipant();
154     ReceiveStream stream = evt.getReceiveStream();
155
156     if (evt instanceof RemotePayloadChangeEvent) {
157
158         info("Received a RTP PayloadChangeEvent");
159         error("Cannot handle payload change");
160         System.exit(1);
161     }
162
163     else if (evt instanceof NewReceiveStreamEvent) {
```

```
164
165             try {
166                 stream = ((NewReceiveStreamEvent)evt).getReceiveStream();
167                 DataSource ds = stream.getDataSource();
168
169                 // Find the format of the new RTP receive stream
170                 RTPControl ctl =
171                     (RTPControl)ds.getControl("javax.media.rtp.RTPControl");
172                 if (ctl != null){
173                     info("Received new RTP stream: " + ctl.getFormat());
174                 } else
175                     info("Received new RTP stream");
176
177                 // Find the CNAME of the participant sending this stream
178                 if (participant == null)
179                     info("The sender of this stream had yet to be identified");
180                 else {
181                     info("The stream comes from: " + participant.getCNAME());
182                 }
183
184                 // Try to create a processor to handle the data coming from
185                 // this new RTP receive stream
186                 Processor processor = null;
187                 try {
188                     processor = javax.media.Manager.createProcessor(ds);
189                 } catch (NoProcessorException npe) {
190                     error("Couldn't create processor");
191                     System.exit(1);
192                 } catch (IOException ioe) {
193                     error("I/O exception while creating processor");
194                     System.exit(1);
195                 }
196
197                 // Wait for it to be in configured state
198                 boolean result = waitForState(processor, Processor.Configured);
199                 if (result == false) {
200                     error("Couldn't configure processor");
201                     System.exit(1);
202                 }
203
204                 // Get the tracks from the processor
205                 TrackControl [] tracks = processor.getTrackControls();
206                 if (tracks == null || tracks.length < 1) {
207                     error("Couldn't find tracks in processor");
208                     System.exit(1);
209                 }
210
211                 Format supported[];
212                 Format chosen;
213                 boolean atLeastOneTrack = false;
214
215                 // Configure the tracks
216                 for (int i = 0; i < tracks.length; i++) {
217                     Format format = tracks[i].getFormat();
218
219                     if (tracks[i].isEnabled()) {
220
221                         supported = tracks[i].getSupportedFormats();
222
223                         if (supported.length > 0) {
224                             if (supported[0] instanceof VideoFormat) {
225                                 chosen = checkForVideoSizes(tracks[i].getFormat(),
226                                                             supported[0]);
227                             } else
228                                 chosen = supported[0];
229                             tracks[i].setFormat(chosen);
230                             info("Track " + i + " is set to transmit as:");
231                             info("  " + chosen);
232                             atLeastOneTrack = true;
233                         } else
234                             tracks[i].setEnabled(false);
235                       } else
236                         tracks[i].setEnabled(false);
237                 }
238
239                 if (!atLeastOneTrack) {
240                     error("Couldn't set any of the tracks to a valid RTP format");
241                     System.exit(1);
242                 }
243
244                 // Realize the processor
245                 result = waitForState(processor, Controller.Realized);
246                 if (result == false) {
```

```
247                     error("Couldn't realize processor");
248                     System.exit(1);
249                 }
250
251                 // Get the output data source of the processor
252                 DataSource pds = processor.getDataOutput();
253
254                 // Create a player with this datasource
255                 Player p = javax.media.Manager.createPlayer(pds);
256                 if (p == null) {
257                     error("Couldn't create player");
258                     System.exit(1);
259                 }
260
261                 p.addControllerListener(this);
262                 p.realize();
263                 PlayerWindow pw = new PlayerWindow(p, stream);
264                 playerWindows.addElement(pw);
265
266                 // Start the processor
267                 processor.start();
268
269                 // Notify intialize() that a new stream had arrived
270                 synchronized (dataSync) {
271                     dataReceived = true;
272                     dataSync.notifyAll();
273                 }
274
275             } catch (Exception e) {
276                 error("New receive stream event exception " + e.getMessage());
277                 System.exit(1);
278             }
279         }
280
281         else if (evt instanceof StreamMappedEvent) {
282
283             if (stream != null && stream.getDataSource() != null) {
284                 DataSource ds = stream.getDataSource();
285                 // Find the CNAME of the participant sending this stream
286                 RTPControl ctl =
287                     (RTPControl)ds.getControl("javax.media.rtp.RTPControl");
288                 info("The previously unidentified stream ");
289                 if (ctl != null)
290                     info("      " + ctl.getFormat());
291                 info("had now been identified as sent by: "
292                      + participant.getCNAME());
293             }
294         }
295
296         else if (evt instanceof TimeoutEvent) {
297             info("" + participant.getCNAME() + " leaved (time out)");
298             PlayerWindow pw = find(stream);
299             if (pw != null) {
300                 pw.close();
301                 playerWindows.removeElement(pw);
302             }
303             System.exit(0);
304         }
305
306         else if (evt instanceof ByeEvent) {
307             info("Got \"bye\" from: " + participant.getCNAME());
308             PlayerWindow pw = find(stream);
309             if (pw != null) {
310                 pw.close();
311                 playerWindows.removeElement(pw);
312             }
313         }
314     }
315
316     // Controller listener for the Players
317     public synchronized void controllerUpdate(ControllerEvent ce) {
318
319         Player p = (Player)ce.getSourceController();
320
321         if (p == null)
322             return;
323
324         if (ce instanceof RealizeCompleteEvent) {
325             PlayerWindow pw = find(p);
326             if (pw == null) {
327                 error("Internal error!");
328                 System.exit(1);
329             }
```

```
330             pw.initialize();
331             pw.setVisible(true);
332             p.start();
333         }
334 
335         if (ce instanceof ControllerErrorEvent) {
336             p.removeControllerListener(this);
337             PlayerWindow pw = find(p);
338             if (pw != null) {
339                 pw.close();
340                 playerWindows.removeElement(pw);
341             }
342             error("Internal error: " + ce);
343         }
344     }
345 
346     private void info(String msg) {
347         System.out.println("[INFO] Client RTP Receiver: " + msg);
348     }
349 
350     private void error(String msg) {
351         System.err.println("[ERROR] Client RTP Receiver: " + msg);
352     }
353 
354     // Check if the video dimensions of original format matches one of the H263
355     // standard video formats. Returns the most adequate format in relation
356     // with the supported format
357     Format checkForVideoSizes(Format original, Format supported) {
358 
359         int width, height;
360         Dimension size = ((VideoFormat)original).getSize();
361         Format h263Fmt = new Format(VideoFormat.H263_RTP);
362 
363         if (supported.matches(h263Fmt)) {
364             // H263 only support specific dimensions
365             if (size.width < 128) {
366                 width = 128;
367                 height = 96;
368             } else if (size.width < 176) {
369                 width = 176;
370                 height = 144;
371             } else {
372                 width = 352;
373                 height = 288;
374             }
375         } else
376             return supported;
377 
378         return (new VideoFormat(null,
379                                 new Dimension(width, height),
380                                 Format.NOT_SPECIFIED,
381                                 null,
382                                 Format.NOT_SPECIFIED)).intersects(supported);
383     }
384 
385     // Convenience methods to handle processor's state changes
386 
387     private Integer stateLock = new Integer(0);
388     private boolean failed = false;
389 
390     Integer getStateLock() {
391         return stateLock;
392     }
393 
394     void setFailed() {
395         failed = true;
396     }
397 
398     private synchronized boolean waitForState(Processor p, int state) {
399         p.addControllerListener(new StateListener());
400         failed = false;
401 
402         // Call the required method on the processor
403         if (state == Processor.Configured) {
404             p.configure();
405         } else if (state == Processor.Realized) {
406             p.realize();
407         }
408 
409         while (p.getState() < state && !failed) {
410             synchronized (getStateLock()) {
411                 try {
412                     getStateLock().wait();
```

```
413                 } catch (InterruptedException ie) {
414                     return false;
415                 }
416             }
417         }
418 
419         if (failed)
420             return false;
421         else
422             return true;
423     }
424 
425     // Internal classes
426 
427     // Convenience class to handle processor's state changes
428     class StateListener implements ControllerListener {
429 
430         public void controllerUpdate(ControllerEvent ce) {
431 
432             // If there was an error during configure or
433             // realize, the processor will be closed
434             if (ce instanceof ControllerClosedEvent)
435                 setFailed();
436 
437             // All controller events, send a notification
438             // to the waiting thread in waitForState method
439             if (ce instanceof ControllerEvent) {
440                 synchronized (getStateLock()) {
441                     getStateLock().notifyAll();
442                 }
443             }
444         }
445     }
446 
447     // GUI classes for the Player.
448     class PlayerWindow extends Frame {
449 
450         Player player;
451         ReceiveStream stream;
452 
453         PlayerWindow(Player p, ReceiveStream strm) {
454             player = p;
455             stream = strm;
456         }
457 
458         public void initialize() {
459             add(new PlayerPanel(player));
460         }
461 
462         public void close() {
463             player.close();
464             setVisible(false);
465             dispose();
466         }
467 
468         public void addNotify() {
469             super.addNotify();
470             pack();
471         }
472     }
473 
474     // GUI classes for the Player.
475     class PlayerPanel extends Panel {
476 
477         Component vc, cc;
478 
479         PlayerPanel(Player p) {
480             setLayout(new BorderLayout());
481             if ((vc = p.getVisualComponent()) != null)
482                 add("Center", vc);
483             if ((cc = p.getControlPanelComponent()) != null)
484                 add("South", cc);
485         }
486 
487         public Dimension getPreferredSize() {
488             int w = 0, h = 0;
489             if (vc != null) {
490                 Dimension size = vc.getPreferredSize();
491                 w = size.width;
492                 h = size.height;
493             }
494             if (cc != null) {
495                 Dimension size = cc.getPreferredSize();
```

```
496                     if (w == 0)
497                         w = size.width;
498                     h += size.height;
499                 }
500                 if (w < 160)
501                     w = 160;
502                 return new Dimension(w, h);
503             }
504         }
505 }
```

Prototype/Client/ClientRTPSocketAdapter.java

This file comes from the sample code provided by Sun Microsystems. It is available in its original form at `http://java.sun.com/products/java-media/jmf/2.1.1/solutions/RTPSocketAdapter.java`.

```
1  /*
2   * @(#)RTPSocketAdapter.java     1.2 01/03/13
3   *
4   * Copyright (c) 1999-2001 Sun Microsystems, Inc. All Rights Reserved.
5   *
6   * Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
7   * modify and redistribute this software in source and binary code form,
8   * provided that i) this copyright notice and license appear on all copies of
9   * the software; and ii) Licensee does not utilize the software in a manner
10  * which is disparaging to Sun.
11  *
12  * This software is provided "AS IS," without a warranty of any kind. ALL
13  * EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY
14  * IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR
15  * NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE
16  * LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING
17  * OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS
18  * LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT,
19  * INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER
20  * CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF
21  * OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE
22  * POSSIBILITY OF SUCH DAMAGES.
23  *
24  * This software is not designed or intended for use in on-line control of
25  * aircraft, air traffic, aircraft navigation or aircraft communications; or in
26  * the design, construction, operation or maintenance of any nuclear
27  * facility. Licensee represents and warrants that it will not use or
28  * redistribute the Software for such purposes.
29  */
30
31 package Client;
32
33 import java.io.IOException;
34 import java.net.InetAddress;
35 import java.net.DatagramSocket;
36 import java.net.MulticastSocket;
37 import java.net.DatagramPacket;
38 import java.net.SocketException;
39
40 import javax.media.protocol.DataSource;
41 import javax.media.protocol.PushSourceStream;
42 import javax.media.protocol.ContentDescriptor;
43 import javax.media.protocol.SourceTransferHandler;
44 import javax.media.rtp.RTPConnector;
45 import javax.media.rtp.OutputDataStream;
46
47
48 /**
49  * An implementation of RTPConnector based on UDP sockets.
50  */
51 public class ClientRTPSocketAdapter implements RTPConnector {
52
53     DatagramSocket dataSock;
54     DatagramSocket ctrlSock;
55
56     InetAddress addr;
57     int port;
58
59     SockInputStream dataInStrm = null, ctrlInStrm = null;
60     SockOutputStream dataOutStrm = null, ctrlOutStrm = null;
61
62
63     public ClientRTPSocketAdapter(InetAddress addr, int port) throws IOException {
64         this(addr, port, 1);
65     }
66
67     public ClientRTPSocketAdapter(InetAddress addr, int port, int ttl)
68         throws IOException {
69
70         try {
71
72             if (addr.isMulticastAddress()) {
73                 dataSock = new MulticastSocket(port);
74                 ctrlSock = new MulticastSocket(port+1);
```

```
75                  ((MulticastSocket)dataSock).joinGroup(addr);
76                  ((MulticastSocket)dataSock).setTimeToLive(ttl);
77                  ((MulticastSocket)ctrlSock).joinGroup(addr);
78                  ((MulticastSocket)ctrlSock).setTimeToLive(ttl);
79              } else {
80                  dataSock = new DatagramSocket(port, InetAddress.getLocalHost());
81                  ctrlSock = new DatagramSocket(port+1, InetAddress.getLocalHost());
82              }
83
84
85          } catch (SocketException e) {
86              throw new IOException(e.getMessage());
87          }
88
89          this.addr = addr;
90          this.port = port;
91      }
92
93      /**
94       * Returns an input stream to receive the RTP data.
95       */
96      public PushSourceStream getDataInputStream() throws IOException {
97          if (dataInStrm == null) {
98              dataInStrm = new SockInputStream(dataSock, addr, port);
99              dataInStrm.start();
100         }
101         return dataInStrm;
102     }
103
104     /**
105      * Returns an output stream to send the RTP data.
106      */
107     public OutputDataStream getDataOutputStream() throws IOException {
108         if (dataOutStrm == null)
109             dataOutStrm = new SockOutputStream(dataSock, addr, port);
110         return dataOutStrm;
111     }
112
113     /**
114      * Returns an input stream to receive the RTCP data.
115      */
116     public PushSourceStream getControlInputStream() throws IOException {
117         if (ctrlInStrm == null) {
118             ctrlInStrm = new SockInputStream(ctrlSock, addr, port+1);
119             ctrlInStrm.start();
120         }
121         return ctrlInStrm;
122     }
123
124     /**
125      * Returns an output stream to send the RTCP data.
126      */
127     public OutputDataStream getControlOutputStream() throws IOException {
128         if (ctrlOutStrm == null)
129             ctrlOutStrm = new SockOutputStream(ctrlSock, addr, port+1);
130         return ctrlOutStrm;
131     }
132
133     /**
134      * Close all the RTP, RTCP streams.
135      */
136     public void close() {
137         if (dataInStrm != null)
138             dataInStrm.kill();
139         if (ctrlInStrm != null)
140             ctrlInStrm.kill();
141         dataSock.close();
142         ctrlSock.close();
143     }
144
145     /**
146      * Set the receive buffer size of the RTP data channel.
147      * This is only a hint to the implementation.  The actual implementation
148      * may not be able to do anything to this.
149      */
150     public void setReceiveBufferSize( int size) throws IOException {
151         dataSock.setReceiveBufferSize(size);
152     }
153
154     /**
155      * Get the receive buffer size set on the RTP data channel.
156      * Return -1 if the receive buffer size is not applicable for
157      * the implementation.
```

```
158         */
159         public int getReceiveBufferSize() {
160             try {
161                 return dataSock.getReceiveBufferSize();
162             } catch (Exception e) {
163                 return -1;
164             }
165         }
166 
167         /**
168          * Set the send buffer size of the RTP data channel.
169          * This is only a hint to the implementation.  The actual implementation
170          * may not be able to do anything to this.
171          */
172         public void setSendBufferSize( int size) throws IOException {
173             dataSock.setSendBufferSize(size);
174         }
175 
176         /**
177          * Get the send buffer size set on the RTP data channel.
178          * Return -1 if the send buffer size is not applicable for
179          * the implementation.
180          */
181         public int getSendBufferSize() {
182             try {
183                 return dataSock.getSendBufferSize();
184             } catch (Exception e) {
185                 return -1;
186             }
187         }
188 
189         /**
190          * Return the RTCP bandwidth fraction.  This value is used to
191          * initialize the RTPManager.  Check RTPManager for more detauls.
192          * Return -1 to use the default values.
193          */
194         public double getRTCPBandwidthFraction() {
195             return -1;
196         }
197 
198         /**
199          * Return the RTCP sender bandwidth fraction.  This value is used to
200          * initialize the RTPManager.  Check RTPManager for more detauls.
201          * Return -1 to use the default values.
202          */
203         public double getRTCPSenderBandwidthFraction() {
204             return -1;
205         }
206 
207 
208         /**
209          * An inner class to implement an OutputDataStream based on UDP sockets.
210          */
211         class SockOutputStream implements OutputDataStream {
212 
213             DatagramSocket sock;
214             InetAddress addr;
215             int port;
216 
217             public SockOutputStream(DatagramSocket sock, InetAddress addr, int port) {
218                 this.sock = sock;
219                 this.addr = addr;
220                 this.port = port;
221             }
222 
223             public int write(byte data[], int offset, int len) {
224                 try {
225                     sock.send(new DatagramPacket(data, offset, len, addr, port));
226                 } catch (Exception e) {
227                     return -1;
228                 }
229                 return len;
230             }
231         }
232 
233 
234         /**
235          * An inner class to implement an PushSourceStream based on UDP sockets.
236          */
237         class SockInputStream extends Thread implements PushSourceStream {
238 
239             DatagramSocket sock;
240             InetAddress addr;
```

```
241         int port;
242         boolean done = false;
243         boolean dataRead = false;
244
245         SourceTransferHandler sth = null;
246
247         public SockInputStream(DatagramSocket sock, InetAddress addr, int port) {
248             this.sock = sock;
249             this.addr = addr;
250             this.port = port;
251         }
252
253         public int read(byte buffer[], int offset, int length) {
254             DatagramPacket p =
255                 new DatagramPacket(buffer, offset, length, addr, port);
256             try {
257                 sock.receive(p);
258             } catch (IOException e) {
259                 return -1;
260             }
261             synchronized (this) {
262                 dataRead = true;
263                 notify();
264             }
265             return p.getLength();
266         }
267
268         public synchronized void start() {
269             super.start();
270             if (sth != null) {
271                 dataRead = true;
272                 notify();
273             }
274         }
275
276         public synchronized void kill() {
277             done = true;
278             notify();
279         }
280
281         public int getMinimumTransferSize() {
282             return 2 * 1024;    // twice the MTU size, just to be safe.
283         }
284
285         public synchronized void setTransferHandler(SourceTransferHandler sth) {
286             this.sth = sth;
287             dataRead = true;
288             notify();
289         }
290
291         // Not applicable.
292         public ContentDescriptor getContentDescriptor() {
293             return null;
294         }
295
296         // Not applicable.
297         public long getContentLength() {
298             return LENGTH_UNKNOWN;
299         }
300
301         // Not applicable.
302         public boolean endOfStream() {
303             return false;
304         }
305
306         // Not applicable.
307         public Object[] getControls() {
308             return new Object[0];
309         }
310
311         // Not applicable.
312         public Object getControl(String type) {
313             return null;
314         }
315
316         /**
317          * Loop and notify the transfer handler of new data.
318          */
319         public void run() {
320             while (!done) {
321
322                 synchronized (this) {
323                     while (!dataRead && !done) {
```

```
324                             try {
325                                 wait();
326                             } catch (InterruptedException e) { }
327                         }
328                         dataRead = false;
329                     }
330
331                     if (sth != null && !done) {
332                         sth.transferData(this);
333                     }
334                 }
335             }
336         }
337 }
```

A.1.2 Proxy server

Prototype/Proxy/ProxyServer.java

```
1  package Proxy;
2
3  import java.io.*;
4  import java.net.*;
5  import java.util.*;
6
7  import javax.media.Format;
8  import javax.media.format.VideoFormat;
9  import javax.media.PlugInManager;
10
11 import Proxy.Plugin.*;
12
13 public class ProxyServer {
14
15     // Port on which the proxy listens for connections
16     private int listeningPort;
17
18     // Socket used for these connections
19     private ServerSocket server;
20
21     // Maximum number of sessions
22     private int maxSession;
23
24     // Vector used to keep the references of running proxy sessions
25     private Vector sessionVect;
26
27     // Base values for RTP sessions ports
28     public static int FromSrcClientRTPPort = 7000;
29     public static int ToSrcClientRTPPort = 7500;
30
31     public static int FromSrcProxyRTPPort = 8000;
32     public static int ToSrcProxyRTPPort = 8500;
33
34     public static int FromDstClientRTPPort = 9000;
35     public static int ToDstClientRTPPort = 9500;
36
37     public static void main(String[] args) {
38         ProxyServer proxyServer = new ProxyServer(args);
39     }
40
41     public ProxyServer(String[] args) {
42
43         info("Proxy server starting ...");
44         // args[0] == listening port
45         // args[1] == maximum number of session
46
47         if (args.length < 2) {
48             prUsage();
49         }
50
51         listeningPort = java.lang.Integer.parseInt(args[0]);
52         maxSession = java.lang.Integer.parseInt(args[1]);
53
54         // Insert the H263 Video Signer plugin into plugin list
55         if(PlugInManager.addPlugIn("Proxy.Plugin.H263VideoSigner",
56                                    // Plugin input format
57                                    new Format [] {
58                                        new VideoFormat(VideoFormat.H263_RTP)
59                                    },
60                                    // Plugin output format
61                                    new Format [] {
62                                        new VideoFormat(VideoFormat.H263_RTP)
63                                    },
64                                    PlugInManager.CODEC))
65             info("H263 Video Signer: Plugin registered");
66         else
67             error("H263 Video Signer: Error while registering");
68
69         // Insert the H263 Video Verifier plugin into plugin list
70         if(PlugInManager.addPlugIn("Proxy.Plugin.H263VideoVerifier",
71                                    new Format [] {                    // Format in
72                                        new VideoFormat(VideoFormat.H263_RTP)
73                                    },
74                                    new Format [] {                    // Format out
75                                        new VideoFormat(VideoFormat.H263_RTP)
76                                    },
77                                    PlugInManager.CODEC))
```

```
78              info("H263 Video Verifier: Plugin registered");
79          else
80              error("H263 Video Verifier: Error while registering");
81
82          sessionVect = new Vector();
83
84          SessionCleaner sessionCleaner = new SessionCleaner(sessionVect);
85          sessionCleaner.start();
86          SessionWatcher sessionWatcher = new SessionWatcher(sessionVect);
87          sessionWatcher.start();
88
89          // Main loop
90          // Wait for connection
91          try {
92              server = new ServerSocket(listeningPort);
93              info("Proxy server waiting for connection ...");
94              while(true) {
95                  if(sessionVect.size() < maxSession) {
96                      sessionVect.add(new ProxySession(server.accept()));
97                  }
98              }
99          }
100         catch (IOException e) {
101             error("I/O exception");
102         }
103     }
104
105     // Print program usage on the standard output
106     void prUsage() {
107         info("Usage: java ProxyServer <listeningPort> <maxSession>");
108         info("     <listeningPort>: network port number "
109                 + "listened for client connection");
110         info("     <maxSession>: maximum number of session at the same time");
111         System.exit(0);
112     }
113
114     private void info(String msg) {
115         System.out.println("[INFO] Proxy Server: " + msg);
116     }
117
118     private void error(String msg) {
119         System.err.println("[ERROR] Proxy Server: " + msg);
120         System.exit(1);
121     }
122
123     // Internal classes
124
125     // This thread periodically removes the terminated sessions
126     class SessionCleaner extends Thread {
127
128         Vector sessionVect;
129
130         public SessionCleaner(Vector sessionVect) {
131             this.sessionVect = sessionVect;
132         }
133
134         public void run() {
135             int i;
136             ProxySession sessionTmp = null;
137
138             while(true) {
139                 for(i=0;i<sessionVect.size();i++) {
140                     sessionTmp = (ProxySession) sessionVect.get(i);
141                     if(!sessionTmp.state) {
142
143                         if(sessionTmp.signer != null) {
144                             sessionTmp.signer.sessionTransmitter.codec.close();
145                             sessionTmp.signer.sessionTransmitter.close();
146                         }
147                         if(sessionTmp.verifier != null) {
148                             sessionTmp.verifier.sessionTransmitter.codec.close();
149                             sessionTmp.verifier.sessionTransmitter.close();
150                         }
151
152                         sessionVect.removeElementAt(i);
153                         info("Session removed (ID: " + sessionTmp.ID + ")");
154                         i--;
155                     }
156                 }
157                 try {
158                     sleep(5000);
159                 }
160                 catch(InterruptedException e) {
```

```
161                     error("Interrupted exception");
162                 }
163             }
164         }
165         private void info(String msg) {
166             System.out.println("[INFO] Proxy server session cleaner: " + msg);
167         }
168 
169         private void error(String msg) {
170             System.err.println("[ERROR] Proxy server session cleaner: " + msg);
171         }
172     }
173 
174     // This thread periodically prints the list of the active sessions
175     class SessionWatcher extends Thread {
176 
177         Vector sessionVect;
178 
179         public SessionWatcher(Vector sessionVect) {
180             this.sessionVect = sessionVect;
181         }
182 
183         public void run() {
184             int i;
185             ProxySession sessionTmp = null;
186 
187             while(true) {
188                 info("" + sessionVect.size() + " active session(s)");
189                 for(i=0;i<sessionVect.size();i++) {
190                     sessionTmp = (ProxySession) sessionVect.get(i);
191                     info("Session ID: " + sessionTmp.ID);
192                     info("    \\ Proxy Ip: " + sessionTmp.proxyIp);
193                     info("    \\ Client Ip: " + sessionTmp.clientIp);
194                 }
195                 try {
196                     sleep(10000);
197                 }
198                 catch(InterruptedException e) {
199                     error("Interrupted exception");
200                 }
201             }
202         }
203         private void info(String msg) {
204             System.out.println("[INFO] Proxy server session watcher: " + msg);
205         }
206 
207         private void error(String msg) {
208             System.err.println("[ERROR] Proxy server session watcher: " + msg);
209         }
210     }
211 }
```

Prototype/Proxy/ProxySession.java

```
1  package Proxy;
2
3  import java.lang.*;
4  import java.util.*;
5  import java.io.*;
6  import java.net.*;
7
8
9  class ProxySession extends Thread {
10
11     // IP address of the adjacent client
12     String proxyIp;
13     // IP address of the adjacent proxy
14     String clientIp;
15
16     // Transmitters used for the two directions of this RTP communication
17     Transmitter signer = null;
18     Transmitter verifier = null;
19
20     // Session ID
21     int ID;
22     // Requested communication type
23     protected String xType = null;
24
25     // False if and only if this proxy session can be removed
26     // by the session cleaner
27     boolean state;
28
29     // Socket used to communicate with the contacting host
30     Socket soc;
31
32     // Constructor
33     ProxySession(Socket socket) throws IOException {
34         this.soc = socket;
35         this.state = true;
36         this.start();
37     }
38
39     public void run() {
40         info("New proxy Session created");
41 // State number 0
42 // This state number refers to figures 4.6 and 4.7
43         OutputStream os = null;
44         ObjectOutput oos = null;
45         InputStream is = null;
46         ObjectInput ois = null;
47
48         try {
49             os = soc.getOutputStream();
50             oos = new ObjectOutputStream(os);
51             is = soc.getInputStream();
52             ois = new ObjectInputStream(is);
53         }
54         catch(IOException e) {
55             error(soc.getInetAddress().getHostName() + ": I/O error");
56             return;
57         }
58
59         // Read the first message containing host type
60         info(soc.getInetAddress().getHostName() + ": Receiving host type");
61
62         Vector dataVect = null;
63
64         try{
65             dataVect = (Vector) ois.readObject();
66         }
67         catch(ClassNotFoundException e) {
68             error(soc.getInetAddress().getHostName() +
69                     ": Class not found exception");
70             return;
71         }
72         catch(IOException e) {
73             error(soc.getInetAddress().getHostName() + ": I/O error");
74             return;
75         }
76
77         if(((String) dataVect.get(0)).equals("Client")) {
78             info("Host is a client");
79
80             int ToSrcClientRTPPort;
```

```
81              int FromSrcClientRTPPort;
82              int ToDstProxyRTPPort;
83              int FromDstProxyRTPPort;
84
85              // Compute session ID
86              ID = ((Object) this).hashCode();
87              xType = (String) dataVect.get(1);
88              String dstProxyIp = (String) dataVect.get(2);
89              int dstProxyPort = ((Integer) dataVect.get(3)).intValue();
90              String dstHostIp = (String) dataVect.get(4);
91              int dstHostPort = ((Integer) dataVect.get(5)).intValue();
92
93              info("Exchange type: " + xType);
94              info("Destination proxy ip: " + dstProxyIp);
95              info("Destination proxy port: " + dstProxyPort);
96              info("Destination host ip: " + dstHostIp);
97              info("Destination host port: " + dstHostPort);
98  // The state numbers of this block refer to figure 4.6
99  // State number 1
100
101             // Connection to destination proxy
102             info("Connecting to detination proxy " + dstProxyIp);
103             Socket socProxy = null;
104
105             OutputStream osProxy = null;
106             ObjectOutput oosProxy = null;
107             InputStream isProxy = null;
108             ObjectInput oisProxy = null;
109
110             try {
111                 socProxy =
112                     new Socket(InetAddress.getByName(dstProxyIp),dstProxyPort);
113                 osProxy = socProxy.getOutputStream();
114                 oosProxy = new ObjectOutputStream(osProxy);
115             }
116             catch(IOException e) {
117                 try {
118                     // Destination proxy unreachable
119                     // Send error message to source client
120                     dataVect = new Vector();
121                     dataVect.add(new String("Error"));
122                     oos.writeObject(dataVect);
123                     oos.flush();
124                 }
125                 catch(IOException f) {
126                     error("I/O error");
127                     return;
128                 }
129                 error("I/O error with " + dstProxyIp);
130
131 // State number 9
132                 return;
133             }
134
135             // Send the destination client parameters to the destination proxy
136             info("Sending destination host parameters to proxy "
137                  + dstProxyIp + " ...");
138             try {
139                 dataVect = new Vector();
140                 dataVect.add(new String("Proxy"));
141                 dataVect.add(new Integer(ID));
142                 dataVect.add(xType);
143                 dataVect.add(dstHostIp);
144                 dataVect.add(new Integer(dstHostPort));
145                 if(xType.endsWith("tx"))
146                     dataVect.add(new Integer(ProxyServer.FromSrcProxyRTPPort));
147                 if(xType.startsWith("rx"))
148                     dataVect.add(new Integer(ProxyServer.ToSrcProxyRTPPort));
149
150                 oosProxy.writeObject(dataVect);
151                 oosProxy.flush();
152
153                 ToDstProxyRTPPort = ProxyServer.FromSrcProxyRTPPort;
154                 FromDstProxyRTPPort = ProxyServer.ToSrcProxyRTPPort;
155                 ProxyServer.FromSrcProxyRTPPort += 2;
156                 ProxyServer.ToSrcProxyRTPPort += 2;
157             }
158             catch(IOException e) {
159                 error("I/O error with " + dstProxyIp);
160                 return;
161             }
162
163 // State number 2
```

```
164             // Read the ACK from the destination proxy
165             try {
166                 isProxy = socProxy.getInputStream();
167                 oisProxy = new ObjectInputStream(isProxy);
168 
169                 dataVect = (Vector) oisProxy.readObject();
170                 if(!((String) dataVect.get(0)).equals("ACK")) {
171 
172 // State number 7
173                     try {
174                         // Send error message to source client
175                         dataVect = new Vector();
176                         dataVect.add(new String("Error"));
177                         oos.writeObject(dataVect);
178                         oos.flush();
179                     }
180                     catch(IOException e) {
181                         error("I/O error");
182                         return;
183                     }
184                     error("Destination proxy didn't answer by an ACK");
185 
186 // State number 9
187                     return;
188                 }
189                 info("ACK received from destination proxy");
190             }
191             catch(ClassNotFoundException e) {
192                 error("Class not found exception");
193                 return;
194             }
195             catch(IOException e) {
196                 error("I/O error with " + dstProxyIp);
197                 return;
198             }
199 
200 // State number 3
201             // Send the RTP session parameters to the connected client
202             try {
203                 dataVect = new Vector();
204                 dataVect.add(new Integer(ID));
205                 if(xType.endsWith("tx"))
206                     dataVect.add(new Integer(ProxyServer.FromSrcClientRTPPort));
207                 if(xType.startsWith("rx"))
208                     dataVect.add(new Integer(ProxyServer.ToSrcClientRTPPort));
209 
210                 oos.writeObject(dataVect);
211                 oos.flush();
212 
213                 info("RTP session parameters sent to client");
214                 FromSrcClientRTPPort = ProxyServer.FromSrcClientRTPPort;
215                 ToSrcClientRTPPort = ProxyServer.ToSrcClientRTPPort;
216                 ProxyServer.FromSrcClientRTPPort += 2;
217                 ProxyServer.ToSrcClientRTPPort += 2;
218             }
219             catch(IOException e) {
220                 try {
221                     dataVect = new Vector();
222                     dataVect.add(new String("Error"));
223 
224                     oosProxy.writeObject(dataVect);
225                     oosProxy.flush();
226                 }
227                 catch(IOException f) {
228                     error("I/O error with " + dstProxyIp);
229                     return;
230                 }
231 
232                 error("I/O error with " + soc.getInetAddress().getHostName());
233 
234 // State number 9
235                 return;
236             }
237 
238 // State number 4
239             // Read the confirmation from the source client
240             try {
241                 dataVect = (Vector) ois.readObject();
242                 String confMsg = (String) dataVect.get(0);
243                 try {
244                     // Send confirmation message to source client
245                     dataVect = new Vector();
246                     if(confMsg.equals("OK"))
```

```
247
248 // State number 5
249                         dataVect.add(confMsg);
250                     else
251
252 // State number 8
253                         dataVect.add(new String("Error"));
254                     oosProxy.writeObject(dataVect);
255                     oosProxy.flush();
256                 }
257                 catch(IOException e) {
258                     error("I/O error with " +  soc.getInetAddress().getHostName());
259                     return;
260                 }
261                 if(!confMsg.equals("OK")) {
262                     error("Error message received from source client");
263
264 // State number 9
265                     return;
266                 }
267             }
268             catch(ClassNotFoundException e) {
269                 error("Class not found exception");
270                 return;
271             }
272             catch(IOException e) {
273                 error("I/O error with " + dstProxyIp);
274                 return;
275             }
276
277 // State number 6
278             try {
279                 oisProxy.close();
280                 oosProxy.close();
281                 socProxy.close();
282             }
283             catch(IOException e) {
284                 error("I/O error");
285                 return;
286             }
287
288             proxyIp = dstProxyIp;
289             clientIp = soc.getInetAddress().getHostName();
290
291             info("RTP Transmission to Destination Proxy on Port: "
292                  + ToDstProxyRTPPort);
293             info("RTP Reception from Destination Proxy on Port: "
294                  + FromDstProxyRTPPort);
295             info("RTP Transmission to Source Client on Port: "
296                  + ToSrcClientRTPPort);
297             info("RTP Reception from Source Client on Port: "
298                  + FromSrcClientRTPPort);
299
300             if(xType.endsWith("tx")) {
301                 // Start stream from source client to destination proxy
302                 signer = new Transmitter(clientIp,FromSrcClientRTPPort,
303                                          proxyIp,ToDstProxyRTPPort,
304                                          "Signer",this);
305                 signer.start();
306                 info("Stream from source client to destination proxy started");
307             }
308
309             if(xType.startsWith("rx")) {
310                 // Start stream from destination proxy to source client
311                 verifier = new Transmitter(proxyIp,FromDstProxyRTPPort,
312                                            clientIp,ToSrcClientRTPPort,
313                                            "Verifier",this);
314                 verifier.start();
315                 info("Stream from destination proxy to source client started");
316             }
317         }
318         else if(((String) dataVect.get(0)).equals("Proxy")) {
319             info("Host is a proxy");
320
321             int ToSrcProxyRTPPort = 0;
322             int FromSrcProxyRTPPort = 0;
323             int ToDstClientRTPPort;
324             int FromDstClientRTPPort;
325
326             ID = ((Integer) dataVect.get(1)).intValue();
327             xType = (String) dataVect.get(2);
328             String dstHostIp = (String) dataVect.get(3);
329             int dstHostPort = ((Integer) dataVect.get(4)).intValue();
```

```
330             int shift = 0;
331             if(xType.endsWith("tx")) {
332                 FromSrcProxyRTPPort = ((Integer) dataVect.get(5)).intValue();
333                 shift = 1;
334             }
335             if(xType.startsWith("rx"))
336                 ToSrcProxyRTPPort = ((Integer) dataVect.get(5+shift)).intValue();
337 
338             info("Session ID: " + ID);
339             info("Exchange type: " + xType);
340             info("Destination host ip: " + dstHostIp);
341             info("Destination host port: " + dstHostPort);
342 // The state numbers of this block refer to figure 4.7
343 // State number 1
344 
345             // Connection to destination host
346             Socket socClient = null;
347 
348             OutputStream osClient = null;
349             ObjectOutput oosClient = null;
350             InputStream isClient = null;
351             ObjectInput oisClient = null;
352 
353             try {
354                 info("Connecting to destination host");
355                 socClient =
356                     new Socket(InetAddress.getByName(dstHostIp), dstHostPort);
357                 osClient = socClient.getOutputStream();
358                 oosClient = new ObjectOutputStream(osClient);
359 
360                 // Send the RTP session parameters to the host
361                 info("Sending the RTP session parameters to the host");
362                 dataVect = new Vector();
363                 dataVect.add(new Integer(ID));
364                 dataVect.add(xType);
365                 if(xType.endsWith("tx"))
366                     dataVect.add(new Integer(ProxyServer.ToDstClientRTPPort));
367                 if(xType.startsWith("rx"))
368                     dataVect.add(new Integer(ProxyServer.FromDstClientRTPPort));
369 
370                 oosClient.writeObject(dataVect);
371                 oosClient.flush();
372 
373                 ToDstClientRTPPort = ProxyServer.ToDstClientRTPPort;
374                 FromDstClientRTPPort = ProxyServer.FromDstClientRTPPort;
375                 ProxyServer.ToDstClientRTPPort += 2;
376                 ProxyServer.FromDstClientRTPPort += 2;
377             }
378             catch(IOException e) {
379                 try {
380                     dataVect = new Vector();
381                     dataVect.add(new String("Error"));
382                     oos.writeObject(dataVect);
383                     oos.flush();
384                 }
385                 catch(IOException f) {
386                     error("I/O error with " + dstHostIp);
387                     return;
388                 }
389 
390                 error("I/O error with " + dstHostIp);
391 
392 // State number 9
393                 return;
394             }
395 
396 // State number 2
397             // Read the ACK from the destination client
398             try {
399                 isClient = socClient.getInputStream();
400                 oisClient = new ObjectInputStream(isClient);
401 
402                 dataVect = (Vector) oisClient.readObject();
403                 if(!((String) dataVect.get(0)).equals("ACK")) {
404 
405 // State number 7
406                     try {
407                         // Send error message to source client
408                         dataVect = new Vector();
409                         dataVect.add(new String("Error"));
410                         oos.writeObject(dataVect);
411                         oos.flush();
412                     }
```

```
413                     catch(IOException e) {
414                         error("I/O error");
415                         return;
416                     }
417                     error("Destination client didn't answer by an ACK");
418 
419 // State number 9
420                     return;
421                 }
422                 info("ACK received from destination client");
423             }
424             catch (ClassNotFoundException e) {
425                 error("Class not found exception");
426                 return;
427             }
428             catch (IOException e) {
429                 error("I/O error");
430                 return;
431             }
432 
433 // State number 3
434             // Send an ACK to the source proxy
435             try {
436                 dataVect = new Vector();
437                 dataVect.add(new String("ACK"));
438                 oos.writeObject(dataVect);
439                 oos.flush();
440                 info("ACK sent to source proxy");
441             }
442             catch (IOException e) {
443                 try {
444                     dataVect = new Vector();
445                     dataVect.add(new String("Error"));
446                     oosClient.writeObject(dataVect);
447                     oosClient.flush();
448                 }
449                 catch(IOException f) {
450                     error("I/O error with " + dstHostIp);
451                     return;
452                 }
453 
454                 error("I/O error with " + soc.getInetAddress().getHostName());
455 
456 // State number 9
457                 return;
458             }
459 
460 // State number 4
461             // Read the confirmation from the source client
462             try {
463                 dataVect = (Vector) ois.readObject();
464                 String confMsg = (String) dataVect.get(0);
465                 try {
466                     // Send confirmation message to source client
467                     dataVect = new Vector();
468                     if(confMsg.equals("OK"))
469 
470 // State number 5
471                         dataVect.add(confMsg);
472                     else
473 
474 // State number 8
475                         dataVect.add(new String("Error"));
476                     oosClient.writeObject(dataVect);
477                     oosClient.flush();
478                 }
479                 catch(IOException e) {
480                     error("I/O error with " +  soc.getInetAddress().getHostName());
481                     return;
482                 }
483                 if(!confMsg.equals("OK")) {
484                     error("Error message received from source proxy");
485 
486 // State number 9
487                     return;
488                 }
489             }
490             catch(ClassNotFoundException e) {
491                 error("Class not found exception");
492                 return;
493             }
494             catch(IOException e) {
495                 error("I/O error with " + dstHostIp);
```

```
496                 return;
497             }
498
499 // State number 6
500             try {
501                 oisClient.close();
502                 oosClient.close();
503                 socClient.close();
504             }
505             catch (IOException e) {
506                 error("I/O error with " + dstHostIp);
507                 return;
508             }
509             proxyIp = soc.getInetAddress().getHostName();
510             clientIp = dstHostIp;
511             info("RTP Transmission to Source Proxy on Port: "
512                  + ToSrcProxyRTPPort);
513             info("RTP Reception from Source Proxy on Port: "
514                  + FromSrcProxyRTPPort);
515             info("RTP Transmission to Destination Client on Port: "
516                  + ToDstClientRTPPort);
517             info("RTP Reception from Destination Client on Port: "
518                  + FromDstClientRTPPort);
519
520             if(xType.endsWith("tx")) {
521                 verifier = new Transmitter(proxyIp,FromSrcProxyRTPPort,
522                                            clientIp,ToDstClientRTPPort,
523                                            "Verifier",this);
524                 verifier.start();
525                 info("Stream from source proxy to destination client started");
526             }
527
528             if(xType.startsWith("rx")) {
529                 signer = new Transmitter(clientIp,FromDstClientRTPPort,
530                                          proxyIp,ToSrcProxyRTPPort,
531                                          "Signer",this);
532                 signer.start();
533                 info("Stream from destination client to source proxy started");
534             }
535         }
536         else {
537             error(soc.getInetAddress().getHostName() + ": Undetermined host type");
538             return;
539         }
540
541         try {
542             ois.close();
543             oos.close();
544             soc.close();
545         }
546         catch (IOException e) {
547             error(soc.getInetAddress().getHostName() + ": I/O error");
548             return;
549         }
550
551         info(soc.getInetAddress().getHostName() + ": Connection closed");
552     }
553
554     private void info(String msg) {
555         System.out.println("[INFO] Proxy session (ID: "+ ID +"): " + msg);
556     }
557
558     private void error(String msg) {
559         System.err.println("[ERROR] Proxy session (ID: "+ ID +"): " + msg);
560         state = false;
561     }
562
563     // Thread used to manage one direction of a RTP communication
564     class Transmitter extends Thread {
565
566         protected ProxyRTPSessionsManager sessionTransmitter;
567         private ProxySession proxySession;
568
569         public Transmitter(String clientIp, int clientPort,
570                            String proxyIp, int proxyPort,
571                            String type, ProxySession proxySession) {
572
573             this.proxySession = proxySession;
574             // Create the transmitter managing the transmission going
575             // from a client to a proxy
576             sessionTransmitter =
577                 new ProxyRTPSessionsManager(clientIp,clientPort,
578                                             proxyIp,proxyPort,
```

```
579                                                 type, proxySession.ID);
580         }
581
582         public void run() {
583
584             if(!sessionTransmitter.initialize()) {
585                 error("Failed to start transmitter");
586             }
587
588             try {
589                 while(!sessionTransmitter.isDone())
590                     Thread.sleep(1000);
591             }
592             catch (java.lang.InterruptedException e) {
593                 error("Interrupted exception");
594             }
595
596             // This proxy session can be removed
597             proxySession.state = false;
598         }
599     }
600 }
```

Prototype/Proxy/ProxyRTPSessionsManager.java

```
1  package Proxy;
2
3  import java.io.*;
4  import java.net.*;
5  import java.util.Vector;
6
7  import javax.media.*;
8  import javax.media.rtp.*;
9  import javax.media.rtp.event.*;
10 import javax.media.rtp.rtcp.*;
11 import javax.media.protocol.*;
12 import javax.media.protocol.DataSource;
13 import javax.media.format.AudioFormat;
14 import javax.media.format.VideoFormat;
15 import javax.media.Format;
16 import javax.media.format.FormatChangeEvent;
17 import javax.media.control.BufferControl;
18 import javax.media.control.TrackControl;
19
20 import Proxy.Plugin.*;
21
22 public class ProxyRTPSessionsManager
23     implements ReceiveStreamListener, SessionListener {
24
25     // RTP managers used to receive and transmit the RTP data
26     RTPManager receptionRTPMgr = null;
27     RTPManager transmissionRTPMgr = null;
28
29     // The plugin implementing the chosen authentication scheme
30     Codec codec = null;
31
32     // Data output of the processor connected to the receiving RTP session
33     private DataSource dataOutput = null;
34
35     // IP address and UDP port from which the RTP data are coming
36     private String ipSrc;
37     private int portSrc;
38
39     // IP address and UDP port to which the RTP data are sent
40     private String ipDst;
41     private int portDst;
42
43     // Type of this session, it is equal to "Verifier" if the RTP data are
44     // coming from another Proxy or equal to "Signer" if the RTP data are
45     // coming from a client
46     private String sessionType;
47
48     // The ID of this session
49     private int sessionID;
50
51     // True if and only if these RTP sessions are terminated
52     private boolean done = false;
53
54     // Convenience variable and object used to manage the RTP data waiting loop
55     boolean dataReceived = false;
56     Object dataSync = new Object();
57
58     // Constructor
59     public ProxyRTPSessionsManager(String ipSrc, int portSrc,
60                                    String ipDst, int portDst,
61                                    String sessionType, int sessionID) {
62         this.ipSrc = ipSrc;
63         this.ipDst = ipDst;
64         this.portSrc = portSrc;
65         this.portDst = portDst;
66         this.sessionType = sessionType;
67         this.sessionID = sessionID;
68     }
69
70     // Create the RTP reception session and wait for incoming data
71     protected boolean initialize() {
72
73         // Open the RTP reception session
74         try {
75             info("Open RTP session for: addr: " + ipSrc + " port: " + portSrc);
76
77             receptionRTPMgr = (RTPManager) RTPManager.newInstance();
78             receptionRTPMgr.addSessionListener(this);
79             receptionRTPMgr.addReceiveStreamListener(this);
80
```

```
81              // Initialize the RTPManager with the RTPSocketAdapter
82              receptionRTPMgr.initialize(
83                              new ProxyRTPSocketAdapter(InetAddress.getByName(ipSrc),
84                                                        portSrc, 1));
85              BufferControl bc =
86                  (BufferControl) receptionRTPMgr.getControl("javax.media.control. ↩
                        BufferControl");
87              if (bc != null)
88                  bc.setBufferLength(1000);
89
90          } catch (Exception e){
91              error("Cannot create the RTP Session: " + e.getMessage());
92              return false;
93          }
94
95          // Wait for data to arrive
96          long then = System.currentTimeMillis();
97          // Wait for a maximum of 300 secs
98          long waitingPeriod = 300000;
99
100         try{
101             synchronized (dataSync) {
102                 while (!dataReceived
103                         && System.currentTimeMillis() - then < waitingPeriod) {
104                     if (!dataReceived)
105                         info("Waiting for RTP data to arrive...");
106                     dataSync.wait(1000);
107                 }
108             }
109         } catch (Exception e) {}
110
111         if (!dataReceived) {
112             info("No RTP data was received.");
113             close();
114             return false;
115         }
116         return true;
117     }
118
119     // Create the RTP transmitter and start it
120     private boolean createTransmitter() {
121
122         SendStream sendStream;
123
124         try {
125             transmissionRTPMgr = RTPManager.newInstance();
126
127             // Initialize the RTPManager with the RTPSocketAdapter
128             transmissionRTPMgr.initialize(
129                             new ProxyRTPSocketAdapter(InetAddress.getByName(ipDst),
130                                                       portDst));
131
132             info("Created transmission RTP session: " + ipDst + " " + portDst);
133
134             sendStream = transmissionRTPMgr.createSendStream(dataOutput,0);
135             sendStream.start();
136
137         } catch (Exception  e) {
138             error(e.getMessage());
139             return false;
140         }
141         return true;
142     }
143
144     // Return the state of this sessions manager
145     public boolean isDone() {
146         return done;
147     }
148
149     // Close the session managers
150     protected void close() {
151
152         // Close the reception RTP manager
153         if (receptionRTPMgr != null) {
154             receptionRTPMgr.removeTargets("Closing reception RTP manager");
155             receptionRTPMgr.dispose();
156             receptionRTPMgr = null;
157         }
158
159         // Close the transmission RTP manager
160         if (transmissionRTPMgr != null) {
161             transmissionRTPMgr.removeTargets("Closing transmission RTP manager");
162             transmissionRTPMgr.dispose();
```

```
163             transmissionRTPMgr = null;
164         }
165 
166         done = true;
167     }
168 
169     // RTP session listener.
170     public synchronized void update(SessionEvent evt) {
171         if (evt instanceof NewParticipantEvent) {
172             Participant p = ((NewParticipantEvent)evt).getParticipant();
173             info("A new participant had just joined: " + p.getCNAME());
174         }
175     }
176 
177     // Receive stream listener
178     public synchronized void update(ReceiveStreamEvent evt) {
179 
180         RTPManager mgr = (RTPManager)evt.getSource();
181         Participant participant = evt.getParticipant();
182         ReceiveStream stream = evt.getReceiveStream();
183 
184         if (evt instanceof RemotePayloadChangeEvent) {
185 
186             info("Received a RTP PayloadChangeEvent");
187             error("Cannot handle payload change");
188             close();
189             return;
190         }
191 
192         else if (evt instanceof NewReceiveStreamEvent) {
193 
194             try {
195                 stream = ((NewReceiveStreamEvent)evt).getReceiveStream();
196                 DataSource ds = stream.getDataSource();
197 
198                 // Find out the formats
199                 RTPControl ctl =
200                     (RTPControl)ds.getControl("javax.media.rtp.RTPControl");
201                 if(ctl != null) {
202                     info("Received new RTP stream: " + ctl.getFormat());
203                 }
204                 else
205                     info("Received new RTP stream");
206 
207                 if(participant == null)
208                     info("The sender of this stream had yet to be identified.");
209                 else {
210                     info("The stream comes from: " + participant.getCNAME());
211                 }
212 
213                 // Try to create a processor to handle the input media locator
214                 Processor processor = null;
215                 try {
216                     processor = javax.media.Manager.createProcessor(ds);
217                 } catch (NoProcessorException npe) {
218                     error("Couldn't create processor");
219                     close();
220                     return;
221                 } catch (IOException ioe) {
222                     error("I/O exception while creating processor");
223                     close();
224                     return;
225                 }
226 
227                 // Wait for it to configure
228                 boolean result = waitForState(processor, Processor.Configured);
229                 if (result == false) {
230                     error("Couldn't configure processor");
231                     close();
232                     return;
233                 }
234 
235                 // Get the tracks from the processor
236                 TrackControl [] tracks = processor.getTrackControls();
237                 if (tracks == null || tracks.length < 1) {
238                     error("Couldn't find tracks in processor");
239                     close();
240                     return;
241                 }
242 
243                 Format supported[];
244                 Format chosen;
245                 boolean atLeastOneTrack = false;
```

```
246
247                 // Program the tracks
248                 for (int i = 0; i < tracks.length; i++) {
249
250                     Format format = tracks[i].getFormat();
251
252                     // Insert the H263 Video plugin in
253                     // the video track plugin chain
254                     if(format instanceof VideoFormat) {
255                         try {
256                             info("Setting codec chain for track " + i + " ...");
257
258                             if(sessionType.equals("Verifier")) {
259                                 codec = new H263VideoVerifier();
260                                 ((H263VideoVerifier) codec).sessionID = sessionID;
261                             }
262                             else
263                                 codec = new H263VideoSigner();
264
265                             tracks[i].setCodecChain(new Codec[] { codec });
266
267                             atLeastOneTrack = true;
268                         }
269                         catch (NotConfiguredError e) {
270                             error("Not Configured Error");
271                             close();
272                             return;
273                         }
274                         catch (UnsupportedPlugInException e) {
275                             error("Unsupported PlugIn Exception");
276                             close();
277                             return;
278                         }
279                     }
280                 }
281
282                 if (!atLeastOneTrack) {
283                     error("Couldn't set any of the tracks to a valid RTP format");
284                     close();
285                     return;
286                 }
287
288                 // Realize the processor
289                 result = waitForState(processor, Controller.Realized);
290                 if (result == false) {
291                     error("Couldn't realize processor");
292                     close();
293                     return;
294                 }
295
296                 // Get the output data source of the processor
297                 dataOutput = processor.getDataOutput();
298
299                 // Create a RTP session to transmit the output of the
300                 // processor to the specified IP address and port number
301                 result = createTransmitter();
302                 if (result == false) {
303                     close();
304                     return;
305                 }
306
307                 // Start the processor
308                 processor.start();
309
310                 // Notify intialize() that a new stream had arrived
311                 synchronized (dataSync) {
312                     dataReceived = true;
313                     dataSync.notifyAll();
314                 }
315
316             } catch (Exception e) {
317                 error("New Receive Stream Event Exception " + e.getMessage());
318                 close();
319                 return;
320             }
321         }
322
323         else if (evt instanceof StreamMappedEvent) {
324
325              if (stream != null && stream.getDataSource() != null) {
326                 DataSource ds = stream.getDataSource();
327                 // Find out the formats.
```

```
328                 RTPControl ctl = (RTPControl)ds.getControl("javax.media.rtp. ↩
                        RTPControl");
329                 info("The previously unidentified stream ");
330                 if (ctl != null)
331                     info("      " + ctl.getFormat());
332                 info("had now been identified as sent by: "
333                      + participant.getCNAME());
334             }
335         }
336 
337         else if (evt instanceof TimeoutEvent) {
338             info("" + participant.getCNAME() + " leaved (time out)");
339             close();
340             return;
341         }
342 
343         else if (evt instanceof ByeEvent) {
344             info("Got \"bye\" from: " + participant.getCNAME());
345             close();
346             return;
347         }
348     }
349 
350     private void info(String msg) {
351         System.out.println("[INFO] Proxy RTP " + sessionType
352                            + " (Session ID: " + sessionID + "): " + msg);
353     }
354 
355     private void error(String msg) {
356         System.err.println("[ERROR] Proxy RTP " + sessionType
357                            + " (Session ID: " + sessionID + "): " + msg);
358     }
359 
360     // Convenience methods to handle processor's state changes
361     private Integer stateLock = new Integer(0);
362     private boolean failed = false;
363 
364     Integer getStateLock() {
365         return stateLock;
366     }
367 
368     void setFailed() {
369         failed = true;
370     }
371 
372     private synchronized boolean waitForState(Processor p, int state) {
373         p.addControllerListener(new StateListener());
374         failed = false;
375 
376         // Call the required method on the processor
377         if (state == Processor.Configured) {
378             p.configure();
379         } else if (state == Processor.Realized) {
380             p.realize();
381         }
382 
383         // Wait until we get an event that confirms the
384         // success of the method, or a failure event.
385         // See StateListener inner class
386         while (p.getState() < state && !failed) {
387             synchronized (getStateLock()) {
388                 try {
389                     getStateLock().wait();
390                 } catch (InterruptedException ie) {
391                     return false;
392                 }
393             }
394         }
395 
396         if (failed)
397             return false;
398         else
399             return true;
400     }
401 
402     // Internal classe
403 
404     // Convenience class to handle processor's state changes
405     class StateListener implements ControllerListener {
406 
407         public void controllerUpdate(ControllerEvent ce) {
408 
409             // If there was an error during configure or
```

```
410                 // realize, the processor will be closed
411                 if (ce instanceof ControllerClosedEvent)
412                     setFailed();
413
414                 // All controller events, send a notification
415                 // to the waiting thread in waitForState method.
416                 if (ce instanceof ControllerEvent) {
417                     synchronized (getStateLock()) {
418                         getStateLock().notifyAll();
419                     }
420                 }
421             }
422         }
423 }
```

Prototype/Proxy/ProxyRTPSocketAdapter.java

The content of this file is the same as the content of the prototype client file `Client/ClientRTPSocketAdapter.java` printed in section A.1.1.

Prototype/Proxy/Plugin/H263VideoSigner.java

```
1  package Proxy.Plugin;
2
3  import java.io.*;
4  import java.util.*;
5  import javax.media.*;
6  import javax.media.format.*;
7  import javax.media.format.AudioFormat;
8  import java.security.*;
9
10 public class H263VideoSigner implements Codec {
11
12     private static String CodecName="H263VideoSigner";
13
14     /** chosen input Format **/
15     protected VideoFormat inputFormat;
16
17     /** chosen output Format **/
18     protected VideoFormat outputFormat;
19
20     /** supported input Formats **/
21     protected Format[] supportedInputFormats=new Format[0];
22
23     /** supported output Formats **/
24     protected Format[] supportedOutputFormats=new Format[0];
25
26     // Buffer number
27     int buffNbr = -1;
28
29     // Block number
30     int blockNbr = -1;
31
32     // Input buffer queue
33     Vector inputBufferQueue = new Vector();
34
35     // Output buffer queue
36     Vector outputBufferQueue = new Vector();
37
38     // Number of buffer per block
39     public int blockSize = 8;
40
41     // Tree degree such that log(treeDeg)(blockSize) == belongs to the int set
42     public int treeDeg = 2;
43
44     // Hash size
45     int hashSize = 16;
46
47     // The hasher
48     MessageDigest digester = null;
49
50     // JCA variables
51     PrivateKey privKey;
52     Signature dsa;
53
54     Random randomEngine = null;
55
56     /** initialize the plugin **/
57     public H263VideoSigner() {
58
59         // Set the input and output formats of this plugin
60         supportedInputFormats = new Format[] {
61             new VideoFormat(VideoFormat.H263_RTP)
62         };
63         supportedOutputFormats = new Format[] {
64             new VideoFormat(VideoFormat.H263_RTP)
65         };
66
67         // Initialize the hasher
68         try {
69             digester = MessageDigest.getInstance("MD5");
70         }
71         catch(NoSuchAlgorithmException e) {
```

```
72              error("No such algorithm exception");
73          }
74
75          // Read the private key
76          File f = new File("./privKey.data");
77
78          FileInputStream fis = null;
79
80          try {
81              fis = new FileInputStream(f);
82          }
83          catch(FileNotFoundException e) {
84              error("File not found exception");
85          }
86
87          try {
88              ObjectInput ois = new ObjectInputStream(fis);
89
90              try {
91                  privKey = (PrivateKey) ois.readObject();
92              }
93              catch(ClassNotFoundException e) {
94                  error("Class not found exception");
95              }
96
97              ois.close();
98
99              info("Private key file read");
100         }
101         catch(IOException e) {
102             error("I/O exception");
103         }
104
105         // Signature initialization
106         try {
107             dsa = Signature.getInstance("SHA1withDSA");
108             dsa.initSign(privKey);
109         }
110         catch(NoSuchAlgorithmException e) {
111             error("No such algorithm exception");
112         }
113         catch(InvalidKeyException e) {
114             error("Invalid key exception");
115         }
116     }
117
118     /** get the resources needed by this plugin **/
119     public void open() throws ResourceUnavailableException {
120     }
121
122     /** free the resources allocated by this plugin **/
123     public void close() {
124     }
125
126     /** reset the plugin **/
127     public void reset() {
128     }
129
130     /** no controls for this simple plugin **/
131     public Object[] getControls() {
132         return (Object[]) new Control[0];
133     }
134
135     /** Return the control based on a control type for the plugin **/
136     public Object getControl(String controlType) {
137         try {
138             Class cls = Class.forName(controlType);
139             Object cs[] = getControls();
140             for (int i = 0; i < cs.length; i++) {
141                 if (cls.isInstance(cs[i]))
142                     return cs[i];
143             }
144             return null;
145         } catch (Exception e) { // no such controlType or such control
146             return null;
147         }
148     }
149
150     /************** format methods *************/
151     /** set the input format **/
152     public Format setInputFormat(Format input) {
153         // the following code assumes valid Format
154         inputFormat = (VideoFormat)input;
```

```
155         return (Format)inputFormat;
156     }
157 
158     /** set the output format **/
159     public Format setOutputFormat(Format output) {
160         // the following code assumes valid Format
161         outputFormat = (VideoFormat)output;
162         return (Format)outputFormat;
163     }
164 
165     /** get the input format **/
166     protected Format getInputFormat() {
167         return inputFormat;
168     }
169 
170     /** get the output format **/
171     protected Format getOutputFormat() {
172         return outputFormat;
173     }
174 
175     /** supported input formats **/
176     public Format [] getSupportedInputFormats() {
177         return supportedInputFormats;
178     }
179 
180     /** output Formats for the selected input format **/
181     public Format [] getSupportedOutputFormats(Format in) {
182         if (in == null)
183             return supportedOutputFormats;
184         else {
185             Format outs[] = new Format[1];
186             outs[0] = in;
187             return outs;
188         }
189     }
190 
191     /** return plugin name **/
192     public String getName() {
193         return CodecName;
194     }
195 
196     /** do the processing **/
197     public int process(Buffer inputBuffer, Buffer outputBuffer){
198 
199         // Compute the buffer number
200         buffNbr++;
201 
202         if(buffNbr < 0)
203             buffNbr = 0;
204 
205         // Get the input buffer info
206         int inLength = inputBuffer.getLength();
207         int inOffset = inputBuffer.getOffset();
208         byte[] inData =
209             getValidData((byte[]) inputBuffer.getData(),inOffset,inLength);
210 
211         // Print some info
212         info("*****************************");
213         info("Buffer #" + buffNbr);
214         info("Input buffer length: " + inLength);
215         info("Format/encoding : " + inputBuffer.getFormat().getEncoding());
216 
217         // Add current buffer and its hash to input buffer queue
218         Vector dataVectTmp = new Vector();
219         dataVectTmp.add(inData);
220         dataVectTmp.add(int2ByteArray(buffNbr));
221         byte[] dataTmp = vect2ByteArray(dataVectTmp);
222 
223         inputBufferQueue.add(new InputBufferQueueItem(dataTmp,computeHash(dataTmp)));
224 
225         info("Input buffer queue size: " + inputBufferQueue.size());
226         info("Output buffer queue size: " + outputBufferQueue.size());
227 
228         if(inputBufferQueue.size() == blockSize) {
229             // Generate an authentication tree from the input buffer queue
230             // and fill the output buffer queue
231             blockNbr++;
232             processBlock();
233             info("Block #" + blockNbr + " processed");
234         }
235 
236         if(outputBufferQueue.size() > 0) {
237             byte[] outData = (byte[]) outputBufferQueue.get(0);
```

```
238             outputBufferQueue.remove(0);
239 
240             // Set the output buffer content and properties
241             outputBuffer.setData(outData);
242             outputBuffer.setFormat(inputBuffer.getFormat());
243             outputBuffer.setLength(outData.length);
244             outputBuffer.setOffset(0);
245 
246             info("Buffer sent");
247             return BUFFER_PROCESSED_OK;
248         }
249         else
250             return OUTPUT_BUFFER_NOT_FILLED;
251     }
252 
253     // Extract the valid video data from the input buffer
254     private byte[] getValidData(byte[] data,int offset,int length) {
255 
256         byte[] tmp = new byte[length];
257         int i;
258 
259         for(i=0;i<length;i++)
260             tmp[i] = data[offset + i];
261 
262         return tmp;
263     }
264 
265     // Utils
266 
267     // Convert an integer to a byte array
268     private byte[] int2ByteArray(int val) {
269         byte[] tmp = new byte[4];
270         tmp[0] = (byte) val;
271         tmp[1] = (byte) (val >> 8);
272         tmp[2] = (byte) (val >> 16);
273         tmp[3] = (byte) (val >> 24);
274         return tmp;
275     }
276 
277     // Convert a vector of byte arrays to a single byte array
278     // by appending the vector components
279     private byte[] vect2ByteArray(Vector vectOutData) {
280 
281         byte[] outData;
282         int vectLength = vectOutData.size();
283 
284         int i,length;
285 
286         for(i=0,length=0;i<vectLength;i++)
287             length = length + ((byte[]) (vectOutData.get(i))).length;
288 
289         outData = new byte[length];
290 
291         if(outData == null)
292             error("outData buffer not allocated");
293 
294         int j,outDataActualLength;
295         byte[] tmp;
296 
297         for(i=0,outDataActualLength=0;i<vectLength;i++) {
298             tmp = (byte[]) vectOutData.get(i);
299             for(j=0;j<tmp.length;j++)
300                 outData[outDataActualLength + j] = tmp[j];
301             outDataActualLength = outDataActualLength + tmp.length;
302         }
303         return outData;
304     }
305 
306     // Create an authentication tree with the buffer present inside the input queue
307     // Create the resulting buffers containing the authentication information and
308     // put them inside the output queue
309     private void processBlock() {
310 
311         int i,j,k;
312         Vector tree = new Vector();
313         Vector lastLevel = new Vector();
314         int lastLevelSize;
315         Vector nextLevel;
316         Vector vectTmp;
317 
318         // Insert the leaves in the tree
319         for(i=0;i<blockSize;i++)
320             lastLevel.add(((InputBufferQueueItem) inputBufferQueue.get(i)).hash);
```

```
321         tree.add(lastLevel);
322
323         // Generate the tree
324         info("Generating tree ...");
325         lastLevelSize = blockSize;
326         while(lastLevelSize > 1) {
327             info("Last level size: " + lastLevelSize);
328             nextLevel = new Vector();
329
330             for(i=0;i<lastLevelSize;i+=treeDeg) {
331                 vectTmp = new Vector();
332                 for(j=0;j<treeDeg;j++)
333                     vectTmp.add((byte[]) lastLevel.get(i+j));
334                 nextLevel.add(computeHash(vect2ByteArray(vectTmp)));
335             }
336             lastLevel = nextLevel;
337             tree.add(lastLevel);
338             lastLevelSize = lastLevel.size();
339         }
340         info("Last level size: " + lastLevelSize);
341         info("Tree generated");
342         info("Generating output buffers ...");
343
344         byte[] blockSig = computeSignature((byte[]) lastLevel.get(0));
345
346         int posInLevel;
347         int branchNbr;
348         int currentPos;
349
350         for(i=0;i<blockSize;i++) {
351             vectTmp = new Vector();
352             // Add video data and buffer number
353             vectTmp.add(((InputBufferQueueItem) inputBufferQueue.get(i)).data);
354             // Add hash list
355             posInLevel = i;
356             branchNbr = posInLevel / treeDeg;
357
358             for(j=0;j<tree.size()-1;j++) {
359                 // Add siblings from level j
360                 for(k=0;k<treeDeg;k++) {
361                     currentPos = (branchNbr * treeDeg) + k;
362                     if(posInLevel != currentPos)
363                         vectTmp.add((byte[]) ((Vector) tree.get(j)).get(currentPos));
364                 }
365
366                 // Jump to father node
367                 posInLevel = posInLevel / treeDeg;
368                 branchNbr = posInLevel / treeDeg;
369             }
370             // Add buffer position in block
371             vectTmp.add(int2ByteArray(i));
372             // Add block number
373             vectTmp.add(int2ByteArray(blockNbr));
374             // Add signature
375             vectTmp.add(blockSig);
376             // Add signature size
377             vectTmp.add(new byte[] {(byte) (blockSig.length & 0xff)});
378
379             outputBufferQueue.add(vect2ByteArray(vectTmp));
380         }
381         info("Output buffers generated");
382
383         // Clean input buffer queue
384         for(i=0;i<blockSize;i++)
385             inputBufferQueue.remove(0);
386     }
387
388     // Return the hash value of data
389     private byte[] computeHash(byte[] data) {
390         return digester.digest(data);
391     }
392
393     // Return the signature associated to data
394     private byte[] computeSignature(byte[] data) {
395         byte[] sig = null;
396
397         try {
398             dsa.update(data);
399             sig = dsa.sign();
400         }
401         catch(SignatureException e) {
402             error("Signature Exception");
403         }
```

```
404             return sig;
405         }
406 
407         private void info(String msg) {
408             System.out.println("[INFO] H263 Video Signer Plugin: " + msg);
409         }
410 
411         private void error(String msg) {
412             System.err.println("[ERROR] H263 Video Signer Plugin: " + msg);
413             System.exit(1);
414         }
415 
416         // Class defining the items contained in the input buffer queue
417         class InputBufferQueueItem {
418 
419             // BuffNbr + Buffer
420             byte[] data;
421 
422             // Buffer hash
423             byte[] hash;
424 
425             public InputBufferQueueItem(byte[] data, byte[] hash) {
426                 this.data = data;
427                 this.hash = hash;
428             }
429         }
430 }
```

Prototype/Proxy/Plugin/H263VideoVerifier.java

```
1  package Proxy.Plugin;
2
3  import java.io.*;
4  import java.util.*;
5  import javax.media.*;
6  import javax.media.format.*;
7  import javax.media.format.AudioFormat;
8  import java.security.*;
9
10 import Proxy.*;
11
12 public class H263VideoVerifier implements Codec {
13
14     private static String CodecName="H263VideoVerifier";
15
16     // The file in which the RTP information are saved and
17     // the streams used to use it
18     private File file = null;
19     private FileOutputStream fileos = null;
20     private ObjectOutputStream oos = null;
21
22     // The session identifier
23     public int sessionID;
24
25     /** chosen input Format **/
26     protected VideoFormat inputFormat;
27
28     /** chosen output Format **/
29     protected VideoFormat outputFormat;
30
31     /** supported input Formats **/
32     protected Format[] supportedInputFormats=new Format[0];
33
34     /** supported output Formats **/
35     protected Format[] supportedOutputFormats=new Format[0];
36
37     // Buffer number
38     int buffNbr;
39
40     // Hash size
41     int hashSize = 16;
42
43     // Number of buffer per block
44     public int blockSize = 8;
45
46     // Tree degree such that log(treeDeg)(blockSize) == belongs to the int set
47     public int treeDeg = 2;
48
49     // Tree height
50     private int treeHeight;
51
52     // Number of hashes in a buffer
53     int hashQuantity;
54
55     // The hasher
56     MessageDigest digester = null;
57
58     // JCA variables
59     PublicKey pubKey;
60     Signature dsa;
61
62     /** initialize the plugin **/
63     public H263VideoVerifier() {
64
65         // Compute some parameters
66         hashQuantity =
67             (treeDeg - 1) *
68             (int) (java.lang.Math.log(blockSize) / java.lang.Math.log(treeDeg));
69         info("Number of hashes contained in input buffers: " + hashQuantity);
70
71         treeHeight =
72             (int) (java.lang.Math.log(blockSize) / java.lang.Math.log(treeDeg));
73         info("Tree height: " + treeHeight);
74
75         // Set the input and output formats of this plugin
76         supportedInputFormats = new Format[] {
77             new VideoFormat(VideoFormat.H263_RTP)
78         };
79         supportedOutputFormats = new Format[] {
80             new VideoFormat(VideoFormat.H263_RTP)
```

```
81          };
82
83          // Initialize the hasher
84          try {
85              digester = MessageDigest.getInstance("MD5");
86          }
87          catch(NoSuchAlgorithmException e) {
88              error("No Such Algorithm Exception");
89          }
90
91          // Read the public key
92          File f = new File("./pubKey.data");
93
94          FileInputStream fis = null;
95
96          try {
97              fis = new FileInputStream(f);
98          }
99          catch(FileNotFoundException e) {
100             error("File Not Found Exception");
101         }
102
103         try {
104             ObjectInput ois = new ObjectInputStream(fis);
105
106             try {
107                 pubKey = (PublicKey) ois.readObject();
108             }
109             catch(ClassNotFoundException e) {
110                 error("Class Not Found Exception");
111             }
112
113             ois.close();
114
115             info("Public Key File read");
116         }
117         catch(IOException e) {
118             error("I/O Exception");
119         }
120
121         // Signature verification initialization
122         try {
123             dsa = Signature.getInstance("SHA1withDSA");
124             dsa.initVerify(pubKey);
125         }
126         catch(NoSuchAlgorithmException e) {
127             error("No Such Algorithm Exception");
128         }
129         catch(InvalidKeyException e) {
130             error("Invalid Key Exception");
131         }
132     }
133
134     /** get the resources needed by this plugin **/
135     public void open() throws ResourceUnavailableException {
136         try {
137             String fileID = (new Integer(sessionID)).toString();
138             file = new File("./session_" + fileID + ".sh263");
139             fileos = new FileOutputStream(file);
140             oos = new ObjectOutputStream(fileos);
141             info("File ./session_" + fileID + ".sh263 created");
142         }
143         catch(IOException e) {
144             error("File I/O Exception while opening session file");
145         }
146     }
147
148     /** free the resources allocated by this plugin **/
149     public void close() {
150         try {
151             oos.close();
152             fileos.close();
153         }
154         catch(IOException e) {
155             error("File I/O Exception while closing");
156         }
157     }
158
159     /** reset the plugin **/
160     public void reset() {
161     }
162
163     /** no controls for this simple plugin **/
```

```
164     public Object[] getControls() {
165         return (Object[]) new Control[0];
166     }
167 
168     /** Return the control based on a control type for the plugin **/
169     public Object getControl(String controlType) {
170         try {
171             Class cls = Class.forName(controlType);
172             Object cs[] = getControls();
173             for (int i = 0; i < cs.length; i++) {
174                 if (cls.isInstance(cs[i]))
175                     return cs[i];
176             }
177             return null;
178         } catch (Exception e) { // no such controlType or such control
179             return null;
180         }
181     }
182 
183     /************** format methods *************/
184     /** set the input format **/
185     public Format setInputFormat(Format input) {
186         // the following code assumes valid Format
187         inputFormat = (VideoFormat)input;
188         return (Format)inputFormat;
189     }
190 
191     /** set the output format **/
192     public Format setOutputFormat(Format output) {
193         // the following code assumes valid Format
194         outputFormat = (VideoFormat)output;
195         return (Format)outputFormat;
196     }
197 
198     /** get the input format **/
199     protected Format getInputFormat() {
200         return inputFormat;
201     }
202 
203     /** get the output format **/
204     protected Format getOutputFormat() {
205         return outputFormat;
206     }
207 
208     /** supported input formats **/
209     public Format [] getSupportedInputFormats() {
210         return supportedInputFormats;
211     }
212 
213     /** output Formats for the selected input format **/
214     public Format [] getSupportedOutputFormats(Format in) {
215         if (in == null)
216             return supportedOutputFormats;
217         else {
218             Format outs[] = new Format[1];
219             outs[0] = in;
220             return outs;
221         }
222     }
223 
224     /** return plugin name **/
225     public String getName() {
226         return CodecName;
227     }
228 
229     /** do the processing **/
230     public int process(Buffer inputBuffer, Buffer outputBuffer){
231 
232         // Get the input buffer info
233         int inLength = inputBuffer.getLength();
234         int inOffset = inputBuffer.getOffset();
235         byte[] inData =
236             getValidData((byte[]) inputBuffer.getData(),inOffset,inLength);
237 
238         // Write this buffer to the file
239         try {
240             oos.writeObject(inData);
241             oos.writeObject(new Integer(inLength));
242             oos.writeObject(new Integer(inOffset));
243             oos.flush();
244         }
245         catch(IOException e) {
246             error("File I/O Exception while writing session file");
```

```
247             }
248
249             // Retrieve the block signature from the buffer
250             int sigSize = getSigSize(inData);
251             byte[] sig = extractSignature(inData,sigSize);
252             // Retrieve the block number to which this buffer belongs
253             int blockNbr = extractBlockNbr(inData,sigSize);
254             // Retrieve the current block position in the current block
255             int posInBlock = extractPosition(inData,sigSize);
256             // Retrieve the buffer number
257             buffNbr = extractBuffNbr(inData,sigSize);
258
259             // Compute the length of the video data contained in this buffer
260             int videoDataLength =
261                 inData.length - 1 - sigSize - 4 - 4 - (hashQuantity * hashSize) - 4;
262
263             // Print some info
264             info("*******************************");
265             info("Buffer #" + buffNbr);
266             info("Input buffer length: " + inLength);
267             info("Format/encoding : " + inputBuffer.getFormat().getEncoding());
268             info("Signature size: " + sigSize);
269             info("Block number: " + blockNbr);
270             info("Position in block: " + posInBlock);
271
272             // Compute current buffer hash
273             int i;
274             byte[] dataTmp = new byte[videoDataLength + 4];
275             for(i=0;i<videoDataLength+4;i++)
276                 dataTmp[i] = inData[i];
277             byte[] buffHash = computeHash(dataTmp);
278
279             // Extract signed portion
280             dataTmp = extractSignedData(inData,videoDataLength+4,posInBlock,buffHash);
281
282             // Verify signature
283             if(verifSignature(dataTmp,sig))
284                 info("Signature verification ok");
285             else
286                 error("Bad signature");
287
288             // Extract the video data from this buffer
289             byte[] outData = extractVideoData(inData,videoDataLength);
290
291             // Set the content and properties of the output buffer
292             outputBuffer.setData(outData);
293             outputBuffer.setFormat(inputBuffer.getFormat());
294             outputBuffer.setLength(outData.length);
295             outputBuffer.setOffset(0);
296
297             return BUFFER_PROCESSED_OK;
298         }
299
300         // Extract the valid video data from the input buffer
301         private byte[] getValidData(byte[] data,int offset,int length) {
302
303             byte[] tmp = new byte[length];
304             int i;
305
306             for(i=0;i<length;i++)
307                 tmp[i] = data[offset + i];
308
309             return tmp;
310         }
311
312         // Extract the buffer number
313         private int extractBuffNbr(byte[] data, int sigSize) {
314
315             byte[] tmp = new byte[4];
316             int i;
317             int offset =
318                 data.length - 1 - sigSize - 4 - 4 - (hashQuantity * hashSize) - 4;
319
320             for(i=0;i<4;i++)
321                 tmp[i] = data[offset + i];
322
323             return byteArray2Int(tmp);
324         }
325
326         // Compute the signature size
327         private int getSigSize(byte[] data) {
328             return (int) (data[data.length - 1]);
329         }
```

```
330
331     // Extract the signature from a buffer
332     private byte[] extractSignature(byte[] data,int sigSize) {
333
334         byte[] sig = new byte[sigSize];
335         int offset = data.length - 1 - sigSize;
336         int i;
337         for(i=0;i<sigSize;i++)
338             sig[i] = data[offset + i];
339         return sig;
340     }
341
342     // Extract the block number of a buffer
343     private int extractBlockNbr(byte[] data,int sigSize) {
344
345         byte[] blockNbr = new byte[4];
346         int offset = data.length - 1 - sigSize - 4;
347         int i;
348         for(i=0;i<4;i++)
349             blockNbr[i] = data[offset + i];
350         return byteArray2Int(blockNbr);
351     }
352
353     // Extract the position of a buffer in a block
354     private int extractPosition(byte[] data,int sigSize) {
355
356         byte[] position = new byte[4];
357         int offset = data.length - 1 - sigSize - 4 - 4;
358         int i;
359         for(i=0;i<4;i++)
360             position[i] = data[offset + i];
361         return byteArray2Int(position);
362     }
363
364     // Convert a byte array to an integer
365     private int byteArray2Int(byte[] val) {
366         int tmp = 0;
367         int i;
368         for (i=3;i>0;i--) {
369             tmp = tmp + ((int) val[i] & (int) 0xff);
370             tmp = tmp << 8;
371         }
372         tmp = tmp + ((int) val[0] & (int) 0xff);
373         return tmp;
374     }
375
376     // Extract the video data from a buffer
377     private byte[] extractVideoData(byte[] data, int length) {
378
379         byte[] tmp = new byte[length];
380         int i;
381
382         for(i=0;i<length;i++)
383             tmp[i] = data[i];
384
385         return tmp;
386     }
387
388     // Compute current buffer hash
389     private byte[] computeHash(byte[] data) {
390         return digester.digest(data);
391     }
392
393     // Extract the signed portion of the buffer
394     private byte[] extractSignedData(byte[] data, int offset,
395                                      int posInBlock, byte[] buffHash) {
396
397         int i,j,k,dataTmpOffset;
398         byte[] dataTmp;
399         byte[] currentHash = buffHash;
400         int posInLevel = posInBlock;
401         int posInBranch = posInBlock % treeDeg;
402
403         for(i=0;i<treeHeight;i++) {
404             dataTmp = new byte[treeDeg * hashSize];
405
406             for(j=0;j<treeDeg;j++) {
407                 dataTmpOffset = j * hashSize;
408                 if(j == posInBranch) {
409                     // Copy currentHash into dataTmp
410                     for(k=0;k<hashSize;k++)
411                         dataTmp[dataTmpOffset + k] = currentHash[k];
412                 }
```

```
413                 else {
414                     // Copy hash from data into dataTmp
415                     for(k=0;k<hashSize;k++)
416                         dataTmp[dataTmpOffset + k] = data[offset + k];
417                     offset+=hashSize;
418                 }
419             }
420 
421             currentHash = computeHash(dataTmp);
422             posInLevel = posInLevel / treeDeg;
423             posInBranch = posInLevel % treeDeg;
424         }
425 
426         return currentHash;
427     }
428 
429     // Verify the signature of a buffer
430     private boolean verifSignature(byte[] data, byte[] sig) {
431 
432         boolean result = false;
433 
434         try {
435             dsa.update(data);
436             result = dsa.verify(sig);
437         }
438         catch(SignatureException e) {
439             error("Signature Exception");
440         }
441         return result;
442     }
443 
444     private void info(String msg) {
445         System.out.println("[INFO] H263 Video Verifier Plugin: " + msg);
446     }
447 
448     private void error(String msg) {
449         System.err.println("[ERROR] H263 Video Verifier Plugin: " + msg);
450         System.exit(1);
451     }
452 }
```

A.1.3 Key pair generator

Prototype/KeyGen/KeyPairGen.java

```
1  package KeyGen;
2
3  import java.io.*;
4  import java.security.*;
5
6  public class KeyPairGen {
7
8      public static void main(String[] args) {
9
10         KeyPairGenerator keyGen;
11         SecureRandom secureRandom;
12         KeyPair keyPair = null;
13
14         if(args.length < 2)
15             error("Usage: java KeyGen.KeyPairGen " +
16                   "private_key_filename public_key_filename");
17
18         // Generate the key pair
19         try {
20             keyGen = KeyPairGenerator.getInstance("DSA");
21             secureRandom = SecureRandom.getInstance("SHA1PRNG","SUN");
22             secureRandom.setSeed(secureRandom.generateSeed(16));
23             keyGen.initialize(1024,secureRandom);
24             keyPair = keyGen.generateKeyPair();
25
26             info("Key pair generated");
27         }
28         catch(NoSuchAlgorithmException e) {
29             error("No such algorithm exception");
30         }
31         catch(NoSuchProviderException e) {
32             error("No such provider exception");
33         }
34
35         // Save the keys in the specified files
36         File fPriv = new File(args[0]);
37         File fPub = new File(args[1]);
38
39         FileOutputStream fosPriv = null;
40         FileOutputStream fosPub = null;
41
42         try {
43             fosPriv = new FileOutputStream(fPriv);
44             fosPub = new FileOutputStream(fPub);
45         }
46         catch(FileNotFoundException e) {
47             error("File not found exception");
48         }
49
50         try {
51             ObjectOutput oosPriv = new ObjectOutputStream(fosPriv);
52             ObjectOutput oosPub = new ObjectOutputStream(fosPub);
53
54             oosPriv.writeObject(keyPair.getPrivate());
55             oosPub.writeObject(keyPair.getPublic());
56             oosPriv.flush();
57             oosPub.flush();
58             oosPriv.close();
59             oosPub.close();
60
61             info("Files written");
62         }
63         catch(IOException e) {
64             error("I/O exception");
65         }
66     }
67
68     private static void info(String msg) {
69         System.out.println("[INFO] Key pair generator: " + msg);
70     }
71
72     private static void error(String msg) {
73         System.err.println("[ERROR] Key pair generator: " + msg);
74         System.exit(1);
75     }
76 }
```

A.1.4 Recordings checker

Prototype/Checker/CheckerMain.java

```
1  package Checker;
2
3  import java.io.*;
4  import java.util.*;
5  import java.security.*;
6
7  class CheckerMain {
8
9      // Buffer number
10     int buffNbr;
11
12     // Some constant
13     int hashSize = 16;
14
15     // Number of buffer per block
16     public int blockSize = 8;
17
18     // Tree degree such that log(treeDeg)(blockSize) == belongs to the int set
19     public int treeDeg = 2;
20
21     // Tree height
22     private int treeHeight;
23
24     // Number of hashes in a buffer
25     int hashQuantity;
26
27     // The hasher
28     MessageDigest digester = null;
29
30     // JCA variables
31     PublicKey pubKey;
32     Signature dsa;
33
34     // The file containing the data to authenticate
35     private File file = null;
36     private FileInputStream fileis = null;
37     private ObjectInputStream ois = null;
38
39     public static void main(String args[]) {
40         if(args.length < 1)
41             prUsage();
42         else {
43             CheckerMain checker = new CheckerMain();
44
45             info("Input file: " + args[0]);
46
47             try {
48                 checker.file = new File(args[0]);
49                 checker.fileis = new FileInputStream(checker.file);
50                 checker.ois = new ObjectInputStream(checker.fileis);
51                 info("File opened");
52             }
53             catch(IOException e) {
54                 error("File I/O exception while opening session file");
55             }
56
57             checker.start();
58         }
59     }
60
61     public CheckerMain() {
62
63         // Compute some parameters
64         hashQuantity =
65             (treeDeg - 1) *
66             (int) (java.lang.Math.log(blockSize) / java.lang.Math.log(treeDeg));
67         info("Number of hashes contained in input buffers: " + hashQuantity);
68
69         treeHeight =
70             (int) (java.lang.Math.log(blockSize) / java.lang.Math.log(treeDeg));
71         info("Tree height: " + treeHeight);
72
73         // Initialize the hasher
74         try {
75             digester = MessageDigest.getInstance("MD5");
76         }
77         catch(NoSuchAlgorithmException e) {
```

```
78              error("No such algorithm exception");
79          }
80
81          // Read the public key
82          File f = new File("./pubKey.data");
83
84          FileInputStream fis = null;
85
86          try {
87              fis = new FileInputStream(f);
88          }
89          catch(FileNotFoundException e) {
90              error("File not found exception");
91          }
92
93          try {
94              ObjectInput oisKey = new ObjectInputStream(fis);
95
96              try {
97                  pubKey = (PublicKey) oisKey.readObject();
98              }
99              catch(ClassNotFoundException e) {
100                 error("Class not found exception");
101             }
102
103             oisKey.close();
104
105             info("Public key file read");
106         }
107         catch(IOException e) {
108             error("I/O Exception");
109         }
110
111         // Signature verification initialization
112         try {
113             dsa = Signature.getInstance("SHA1withDSA");
114             dsa.initVerify(pubKey);
115         }
116         catch(NoSuchAlgorithmException e) {
117             error("No such algorithm exception");
118         }
119         catch(InvalidKeyException e) {
120             error("Invalid key exception");
121         }
122     }
123
124     // Start the authentication process
125     void start() {
126
127         byte[] inData = null;
128         int inLength = 0;
129         int inOffset = 0;
130
131         while(true) {
132             // Read a buffer from the data file
133             try {
134                 inData = (byte[]) ois.readObject();
135                 inLength = ((Integer) ois.readObject()).intValue();
136                 inOffset = ((Integer) ois.readObject()).intValue();
137             }
138             catch(ClassNotFoundException e) {
139                 error("Class not found exception");
140             }
141             catch(EOFException e) {
142                 info("File read");
143                 info("File authenticated");
144                 System.exit(0);
145             }
146             catch(IOException e) {
147                 error("I/O exception while reading input file");
148             }
149
150             // Retrieve the signature from the buffer
151             int sigSize = getSigSize(inData);
152             byte[] sig = extractSignature(inData,sigSize);
153
154             // Retrieve the block number
155             int blockNbr = extractBlockNbr(inData,sigSize);
156
157             // Retrieve the position of the current buffer
158             // in the current block
159             int posInBlock = extractPosition(inData,sigSize);
160
```

```
161             // Retrieve the buffer number
162             buffNbr = extractBuffNbr(inData,sigSize);
163
164             // Compute the length of the video data contained in this buffer
165             int videoDataLength =
166                 inData.length - 1 - sigSize - 4 - 4 - (hashQuantity * hashSize) - 4;
167
168             // Print some info
169             info("*****************************");
170             info("Buffer #" + buffNbr);
171             info("Input buffer length: " + inLength);
172             info("Signature size: " + sigSize);
173             info("Block number: " + blockNbr);
174             info("Position in block: " + posInBlock);
175
176             // Compute current buffer hash
177             int i;
178             byte[] dataTmp = new byte[videoDataLength + 4];
179             for(i=0;i<videoDataLength+4;i++)
180                 dataTmp[i] = inData[i];
181             byte[] buffHash = computeHash(dataTmp);
182
183             // Extract signed portion
184             dataTmp =
185                 extractSignedData(inData,videoDataLength+4,posInBlock,buffHash);
186
187             // Verify signature
188             if(verifSignature(dataTmp,sig))
189                 info("Signature verification ok");
190             else
191                 error("********** Bad signature");
192         }
193     }
194
195     // Compute the signature size
196     private int getSigSize(byte[] data) {
197         return (int) (data[data.length - 1]);
198     }
199
200     // Extract the signature from a buffer
201     private byte[] extractSignature(byte[] data,int sigSize) {
202
203         byte[] sig = new byte[sigSize];
204         int offset = data.length - 1 - sigSize;
205         int i;
206         for(i=0;i<sigSize;i++)
207             sig[i] = data[offset + i];
208         return sig;
209     }
210
211     // Extract the block number of a buffer
212     private int extractBlockNbr(byte[] data,int sigSize) {
213
214         byte[] blockNbr = new byte[4];
215         int offset = data.length - 1 - sigSize - 4;
216         int i;
217         for(i=0;i<4;i++)
218             blockNbr[i] = data[offset + i];
219         return byteArray2Int(blockNbr);
220     }
221
222     // Extract the position of a buffer in a block
223     private int extractPosition(byte[] data,int sigSize) {
224
225         byte[] position = new byte[4];
226         int offset = data.length - 1 - sigSize - 4 - 4;
227         int i;
228         for(i=0;i<4;i++)
229             position[i] = data[offset + i];
230         return byteArray2Int(position);
231     }
232
233     // Extract the buffer number
234     private int extractBuffNbr(byte[] data, int sigSize) {
235
236         byte[] tmp = new byte[4];
237         int i;
238         int offset =
239             data.length - 1 - sigSize - 4 - 4 - (hashQuantity * hashSize) - 4;
240
241         for(i=0;i<4;i++)
242             tmp[i] = data[offset + i];
243
```

```
244         return byteArray2Int(tmp);
245     }
246
247     // Compute current buffer hash
248     private byte[] computeHash(byte[] data) {
249         return digester.digest(data);
250     }
251
252     // Extract the signed portion of the buffer
253     private byte[] extractSignedData(byte[] data, int offset,
254                                      int posInBlock, byte[] buffHash) {
255
256         int i,j,k,dataTmpOffset;
257         byte[] dataTmp;
258         byte[] currentHash = buffHash;
259         int posInLevel = posInBlock;
260         int posInBranch = posInBlock % treeDeg;
261
262         for(i=0;i<treeHeight;i++) {
263             dataTmp = new byte[treeDeg * hashSize];
264
265             for(j=0;j<treeDeg;j++) {
266                 dataTmpOffset = j * hashSize;
267                 if(j == posInBranch) {
268                     // Copy currentHash into dataTmp
269                     for(k=0;k<hashSize;k++)
270                         dataTmp[dataTmpOffset + k] = currentHash[k];
271                 }
272                 else {
273                     // Copy hash from data into dataTmp
274                     for(k=0;k<hashSize;k++)
275                         dataTmp[dataTmpOffset + k] = data[offset + k];
276                     offset+=hashSize;
277                 }
278             }
279
280             currentHash = computeHash(dataTmp);
281             posInLevel = posInLevel / treeDeg;
282             posInBranch = posInLevel % treeDeg;
283         }
284
285         return currentHash;
286     }
287
288     // Verify the signature of a buffer
289     private boolean verifSignature(byte[] data, byte[] sig) {
290
291         boolean result = false;
292
293         try {
294             dsa.update(data);
295             result = dsa.verify(sig);
296         }
297         catch(SignatureException e) {
298             error("Signature Exception");
299         }
300         return result;
301     }
302
303     // Convert a byte array to an integer
304     private int byteArray2Int(byte[] val) {
305         int tmp = 0;
306         int i;
307         for (i=3;i>0;i--) {
308             tmp = tmp + ((int) val[i] & (int) 0xff);
309             tmp = tmp << 8;
310         }
311         tmp = tmp + ((int) val[0] & (int) 0xff);
312         return tmp;
313     }
314
315     // Print program usage to standard output
316     static void prUsage() {
317         info("Usage:");
318         info("java Checker.CheckerMain <fileName>");
319         info("     <fileName>: the record of a session");
320
321         System.exit(0);
322     }
323
324     static private void info(String msg) {
325         System.out.println("[INFO] Checker: " + msg);
326     }
```

```
327
328     static private void error(String msg) {
329         System.err.println("[ERROR] Checker: " + msg);
330         System.exit(1);
331     }
332 }
```

A.2 Simulator

A.2.1 Hashing speed

```
1  package HashEval;
2
3  import java.io.*;
4  import java.security.*;
5  import java.util.*;
6
7  public class HashSpeedEval {
8
9      public static void main(String[] args) {
10
11         if(args.length < 2) {
12             error("Usage: java HashEval.HashSpeedEval hashing_algorithm data_file");
13         }
14
15         // The hasher
16         MessageDigest digester = null;
17
18         // The data to hash
19         byte[] data = null;
20
21         // Initialize the hasher
22         try {
23             digester = MessageDigest.getInstance(args[0]);
24             System.out.println("Algorithm: " + digester.getAlgorithm());
25         }
26         catch(NoSuchAlgorithmException e) {
27             error("No Such Algorithm Exception");
28         }
29
30         // Initialize data samples reading from file
31         File f = new File(args[1]);
32
33         FileInputStream fis = null;
34         ObjectInput ois = null;
35
36         try {
37             fis = new FileInputStream(f);
38             ois = new ObjectInputStream(fis);
39         }
40         catch(FileNotFoundException e) {
41             error("File Not Found Exception");
42         }
43         catch(IOException e) {
44             error("I/O Exception");
45         }
46
47         int i = 0;
48         byte[] hash = null;
49
50         // Perform 5000 hash value computations
51         while(i < 5000) {
52
53             try {
54                 data = (byte[]) ois.readObject();
55             }
56             catch(ClassNotFoundException e) {
57                 error("Class Not Found Exception");
58             }
59             catch(IOException e) {
60                 error("I/O Exception");
61             }
62
63             digester.reset();
64             digester.update(data);
65             hash = digester.digest();
66
67             i++;
68         }
69
70         try {
71             ois.close();
72         }
73         catch(IOException e) {
74             error("I/O Exception");
75         }
76
```

```
77          System.out.println("Hash length: " + hash.length);
78
79      }
80
81      private static void info(String msg) {
82          System.out.println("[INFO] Hash speed evaluator: " + msg);
83      }
84
85      private static void error(String msg) {
86          System.err.println("[ERROR] Hash speed evaluator: " + msg);
87          System.exit(1);
88      }
89  }
```

A.2.2 Signing speed

```
1  package SigEval;
2
3  import java.io.*;
4  import java.security.*;
5  import java.util.*;
6
7  public class SigSpeedEval {
8
9      public static void main(String[] args) {
10
11         // JCA variables
12         PrivateKey privKey = null;
13         PublicKey pubKey = null;
14
15         Signature dsa = null;
16
17         // Read the public and private keys
18         File fPriv = new File("./privKey.data");
19         File fPub = new File("./pubKey.data");
20
21         FileInputStream fisPriv = null;
22         FileInputStream fisPub = null;
23         try {
24             fisPriv = new FileInputStream(fPriv);
25             fisPub = new FileInputStream(fPub);
26         }
27         catch(FileNotFoundException e) {
28             error("File not found exception");
29         }
30
31         try {
32             ObjectInput oisPriv = new ObjectInputStream(fisPriv);
33             ObjectInput oisPub = new ObjectInputStream(fisPub);
34
35             try {
36                 privKey = (PrivateKey) oisPriv.readObject();
37                 pubKey = (PublicKey) oisPub.readObject();
38             }
39             catch(ClassNotFoundException e) {
40                 error("Class not found exception");
41             }
42
43             oisPriv.close();
44             oisPub.close();
45
46             info("Public and private key files read");
47         }
48         catch(IOException e) {
49             error("I/O exception");
50         }
51
52         // Signer initialization
53         try {
54             dsa = Signature.getInstance("SHA1withRSA");
55         }
56         catch(NoSuchAlgorithmException e) {
57             error("No such algorithm exception");
58         }
59
60         // Read data samples from file
61         File f = new File("./rec50000.data");
62
63         FileInputStream fis = null;
64
65         try {
66             fis = new FileInputStream(f);
67         }
68         catch(FileNotFoundException e) {
69             error("File Not Found Exception");
70         }
71
72         ObjectInput ois = null;
73         try {
74             ois = new ObjectInputStream(fis);
75         }
76         catch(IOException e) {
77             error("I/O Exception");
78         }
79
80         int i = 0;
```

```
81          byte[] sig = null;
82          byte[] data = null;
83
84          // Perform 5000 signature computations and verifications
85          while(i < 5000) {
86
87              // Signature application initialization
88              try {
89                  dsa.initSign(privKey);
90              }
91              catch(InvalidKeyException e) {
92                  error("Invalid key exception");
93              }
94
95              try {
96                  data = (byte[]) ois.readObject();
97              }
98              catch(ClassNotFoundException e) {
99                  error("Class Not Found Exception");
100             }
101             catch(IOException e) {
102                 error("I/O Exception");
103             }
104             try {
105                 dsa.update(data);
106             }
107             catch(SignatureException e) {
108                 error("Signature Exception");
109             }
110
111             // Signature computation
112             try {
113                 sig = dsa.sign();
114             }
115             catch(SignatureException e) {
116                 error("Signature Exception");
117             }
118
119             // Signature verification initialization
120             try {
121                 dsa.initVerify(pubKey);
122             }
123             catch(InvalidKeyException e) {
124                 error("Invalid key exception");
125             }
126
127             try {
128                 dsa.update(data);
129             }
130             catch(SignatureException e) {
131                 error("Signature Exception");
132             }
133
134             boolean result = false;
135
136             try {
137                 result = dsa.verify(sig);
138             }
139             catch(SignatureException e) {
140                 error("Signature Exception");
141             }
142
143             i++;
144         }
145
146         try {
147             ois.close();
148         }
149         catch(IOException e) {
150             error("I/O Exception");
151         }
152     }
153
154     private static void info(String msg) {
155         System.out.println("[INFO] Signature speed evaluator: " + msg);
156     }
157
158     private static void error(String msg) {
159         System.err.println("[ERROR] Signature speed evaluator: " + msg);
160         System.exit(1);
161     }
162 }
```

A.2.3 Client

Simulator/Client/ClientMain.java

```
1  package Client;
2
3  import java.lang.*;
4  import java.io.*;
5  import java.net.*;
6  import java.util.*;
7  import javax.media.Format;
8  import javax.media.format.VideoFormat;
9  import javax.media.PlugInManager;
10
11 public class ClientMain {
12
13     public static void main(String args[]) {
14
15         if(args.length < 3)
16             prUsage();
17         else {
18
19             String pluginType = "Client.Plugin.H263VideoSigner" + args[2];
20
21             // Insert the specified H263 Video Signer plugin into plugin list
22             if(PlugInManager.addPlugIn(pluginType,
23                                        // Plugin input format
24                                        new Format [] {
25                                            new VideoFormat(VideoFormat.H263_RTP)
26                                        },
27                                        // Plugin output format
28                                        new Format [] {
29                                            new VideoFormat(VideoFormat.H263_RTP)
30                                        },
31                                        PlugInManager.CODEC))
32                 info("H263 Video Signer <" + args[2] + ">: Plugin registered");
33             else
34                 error("H263 Video Signer: Error while registering "
35                       + args[2] + " plugin");
36
37             // Create and start the RTP transmitter
38             ClientRTPTransmitter transmitter =
39                 new ClientRTPTransmitter(args[0], Integer.parseInt(args[1]), args[2]);
40
41             info("Starting RTP transmission ...");
42             String result = transmitter.start();
43
44             if (result != null) {
45                 error(result);
46             }
47             info("Transmission OK");
48
49             // Wait for a user interruption
50             try {
51                 BufferedReader d =
52                     new BufferedReader(new InputStreamReader(System.in));
53                 String line;
54
55                 while((line = d.readLine()) != null) {
56                 }
57                 if(line == null) {
58                     info("Ctrl-d key pressed");
59                     info("Stopping RTP transmitter");
60                     transmitter.stop();
61                 }
62                 d.close();
63             }
64             catch(IOException e) {
65                 error("I/O exception on standard input");
66             }
67             System.exit(0);
68         }
69     }
70
71     // Print the program usage on standard output
72     static void prUsage() {
73         info("Usage:");
74         info("java Client.ClientMain <serverIp> <serverRTPPort> <pluginType>");
75         info("     <serverIp>: server IP address");
76         info("     <serverRTPPort>: server RTP reception port");
77         info("     <pluginType>: plugin short name");
```

```
78              System.exit(0);
79          }
80
81          static private void info(String msg) {
82              System.out.println("[INFO] Client: " + msg);
83          }
84
85          static private void error(String msg) {
86              System.err.println("[ERROR] Client: " + msg);
87              System.exit(1);
88          }
89      }
```

Simulator/Client/ClientRTPTransmitter.java

```
1  package Client;
2
3  import java.awt.*;
4  import java.io.*;
5  import java.util.*;
6  import java.net.InetAddress;
7  import javax.media.*;
8  import javax.media.protocol.*;
9  import javax.media.protocol.DataSource;
10 import javax.media.format.*;
11 import javax.media.control.*;
12 import javax.media.control.TrackControl;
13 import javax.media.control.QualityControl;
14 import javax.media.rtp.*;
15 import javax.media.rtp.rtcp.*;
16 import com.sun.media.rtp.*;
17
18 import Client.Plugin.*;
19
20 public class ClientRTPTransmitter {
21
22     // IP address and UDP port of the destination
23     private String ipDst;
24     private int portDst;
25
26     // Plugin type
27     private String pluginType;
28
29     // The processor sending the data coming from the capture device
30     // to the RTP manager
31     private Processor processor = null;
32
33     // The data output of the processor
34     private DataSource dataOutput = null;
35
36     // The RTP manager used to transmit the data
37     private RTPManager rtpMgr = null;
38
39     // Constructor
40     public ClientRTPTransmitter(String ipDst, int portDst, String pluginType) {
41         this.ipDst = ipDst;
42         this.portDst = portDst;
43         this.pluginType = pluginType;
44     }
45
46     // Start the transmission, return null if transmission starting is successful
47     // Otherwise it returns a string with the reason why the setup failed
48     public synchronized String start() {
49         String result;
50
51         // Create a processor for the specified media locator
52         // and program it to output H263/RTP
53         result = createProcessor();
54         if (result != null)
55             return result;
56
57         // Create a RTP session to transmit the output of the
58         // processor to the specified IP address ipDst and port number portDst
59         result = createTransmitter();
60         if (result != null) {
61             processor.close();
62             processor = null;
63             return result;
64         }
65
66         // Start the transmission
67         processor.start();
68
69         return null;
70     }
71
72     // Stops the transmission if already started
73     public void stop() {
74         synchronized (this) {
75             if (processor != null) {
76                 processor.stop();
77                 processor.close();
78                 processor = null;
79                 rtpMgr.removeTargets("Session ended");
80                 rtpMgr.dispose();
```

```
81              }
82          }
83      }
84
85      // Create the processor receiving the data from the capture device
86      private String createProcessor() {
87          DataSource ds;
88          DataSource clone;
89
90          try {
91              ds = createDataSource(new VideoFormat("RGB",
92                                    new Dimension(352, 288),
93                                    Format.NOT_SPECIFIED,
94                                    null,
95                                    Format.NOT_SPECIFIED));
96          }
97          catch(Exception e) {
98              return "Couldn't create DataSource";
99          }
100
101         // Try to create a processor to handle the input media locator
102         try {
103             processor = javax.media.Manager.createProcessor(ds);
104         }
105         catch(NoProcessorException npe) {
106             return "Couldn't create processor";
107         }
108         catch(IOException ioe) {
109             return "I/O exception creating processor";
110         }
111
112         // Wait for it to configure
113         boolean result = waitForState(processor, Processor.Configured);
114         if (result == false)
115             return "Couldn't configure processor";
116
117         // Get the tracks from the processor
118         TrackControl [] tracks = processor.getTrackControls();
119         if (tracks == null || tracks.length < 1)
120             return "Couldn't find tracks in processor";
121
122         // Set the output content descriptor to RAW_RTP
123         // This will limit the supported formats reported from
124         // Track.getSupportedFormats to only valid RTP formats
125         ContentDescriptor cd = new ContentDescriptor(ContentDescriptor.RAW_RTP);
126         processor.setContentDescriptor(cd);
127
128         Format supported[];
129         Format chosen;
130         boolean atLeastOneTrack = false;
131
132         // Program the tracks
133         for (int i = 0; i < tracks.length; i++) {
134             Format format = tracks[i].getFormat();
135
136             Codec plugin = null;
137
138             // Select the specified plugin
139             if(pluginType.equals("Basic"))
140                 plugin = new H263VideoSignerBasic();
141             else if(pluginType.equals("Simple"))
142                 plugin = new H263VideoSignerSimple();
143             else if(pluginType.equals("EMSS"))
144                 plugin = new H263VideoSignerEMSS();
145             else if(pluginType.equals("Star"))
146                 plugin = new H263VideoSignerStar();
147             else if(pluginType.equals("Tree"))
148                 plugin = new H263VideoSignerTree();
149             else
150                 error("Bad plugin type");
151
152             try {
153                 info("Setting encoding plugins chain for track " + i + " ...");
154                 tracks[i].setCodecChain(
155                       new Codec[] {
156                           new com.sun.media.codec.video.colorspace.JavaRGBToYUV(),
157                           new com.ibm.media.codec.video.h263.NativeEncoder(),
158                           plugin
159                       });
160             }
161             catch (NotConfiguredError e) {
162                 error("Not Configured Error");
163             }
```

```
164             catch (UnsupportedPlugInException e) {
165                 error("Unsupported PlugIn Exception");
166             }
167 
168             if (tracks[i].isEnabled()) {
169 
170                 supported = tracks[i].getSupportedFormats();
171 
172                 if (supported.length > 0) {
173                     if (supported[0] instanceof VideoFormat) {
174                         chosen = checkForVideoSizes(tracks[i].getFormat(),
175                                                     supported[0]);
176                     } else
177                         chosen = supported[0];
178                     tracks[i].setFormat(chosen);
179                     info("Track " + i + " is set to transmit as:");
180                     info("  " + chosen);
181                     atLeastOneTrack = true;
182                 } else
183                     tracks[i].setEnabled(false);
184             } else
185                 tracks[i].setEnabled(false);
186         }
187 
188         if (!atLeastOneTrack)
189             return "Couldn't set any of the tracks to a valid RTP format";
190 
191         // Realize the processor
192         result = waitForState(processor, Controller.Realized);
193         if (result == false)
194             return "Couldn't realize processor";
195 
196         // Get the output data source of the processor
197         dataOutput = processor.getDataOutput();
198 
199         return null;
200     }
201 
202     // Create a data source connected to the capture device
203     DataSource createDataSource(Format format) throws Exception {
204         DataSource ds;
205         Vector devices;
206         CaptureDeviceInfo cdi;
207         MediaLocator ml;
208 
209         // Find devices for format
210         devices = CaptureDeviceManager.getDeviceList(format);
211         if (devices.size() < 1) {
212             error("No Devices for " + format);
213             throw new Exception();
214         }
215         // Pick the first device
216         cdi = (CaptureDeviceInfo) devices.elementAt(0);
217 
218         ml = cdi.getLocator();
219 
220         ds = Manager.createDataSource(ml);
221         ds.connect();
222         if(ds instanceof CaptureDevice)
223             setCaptureFormat((CaptureDevice) ds, format);
224         return ds;
225     }
226 
227     // Set the format of the data coming out of the capture device
228     void setCaptureFormat(CaptureDevice cdev, Format format) {
229         FormatControl [] fcs = cdev.getFormatControls();
230         if (fcs.length < 1)
231             return;
232         FormatControl fc = fcs[0];
233         Format [] formats = fc.getSupportedFormats();
234 
235         for (int i = 0; i < formats.length; i++) {
236             if (formats[i].matches(format)) {
237                 format = formats[i].intersects(format);
238                 info("Setting format " + format);
239                 fc.setFormat(format);
240                 break;
241             }
242         }
243     }
244 
245     // Create and start the RTP transmitter
246     private String createTransmitter() {
```

```
247         SendStream sendStream;
248
249         try {
250             rtpMgr = RTPManager.newInstance();
251
252             // Initialize the RTPManager with the RTPSocketAdapter
253             rtpMgr.initialize(
254                         new ClientRTPSocketAdapter(
255                                                 InetAddress.getByName(ipDst),
256                                                 portDst));
257
258             info("Created RTP session: " + ipDst + " " + portDst);
259
260             sendStream = rtpMgr.createSendStream(dataOutput,0);
261             sendStream.start();
262         } catch (Exception  e) {
263             return e.getMessage();
264         }
265         return null;
266     }
267
268
269
270     // H263 only works for particular sizes
271     Format checkForVideoSizes(Format original, Format supported) {
272
273         int width, height;
274         Dimension size = ((VideoFormat)original).getSize();
275         Format jpegFmt = new Format(VideoFormat.JPEG_RTP);
276         Format h263Fmt = new Format(VideoFormat.H263_RTP);
277
278         if(supported.matches(h263Fmt)) {
279             // H263 only supports specific dimensions
280             if (size.width < 128) {
281                 width = 128;
282                 height = 96;
283             } else if (size.width < 176) {
284                 width = 176;
285                 height = 144;
286             } else {
287                 width = 352;
288                 height = 288;
289             }
290         } else
291             return supported;
292
293         return (new VideoFormat(null,
294                                 new Dimension(width, height),
295                                 Format.NOT_SPECIFIED,
296                                 null,
297                                 Format.NOT_SPECIFIED)).intersects(supported);
298     }
299
300     // Convenience methods used to handle processor's state changes
301     private Integer stateLock = new Integer(0);
302     private boolean failed = false;
303
304     Integer getStateLock() {
305         return stateLock;
306     }
307
308     void setFailed() {
309         failed = true;
310     }
311
312     private synchronized boolean waitForState(Processor p, int state) {
313         p.addControllerListener(new StateListener());
314         failed = false;
315
316         // Call the required method on the processor
317         if (state == Processor.Configured) {
318             p.configure();
319         } else if (state == Processor.Realized) {
320             p.realize();
321         }
322         while (p.getState() < state && !failed) {
323             synchronized (getStateLock()) {
324                 try {
325                     getStateLock().wait();
326                 } catch (InterruptedException ie) {
327                     return false;
328                 }
329             }
```

```
330             }
331
332             if (failed)
333                 return false;
334             else
335                 return true;
336         }
337
338         private void info(String msg) {
339             System.out.println("[INFO] Client RTP Transmitter: " + msg);
340         }
341
342         private void error(String msg) {
343             System.err.println("[ERROR] Client RTP Transmitter: " + msg);
344         }
345
346         // Convenience class used to handle processor's state changes
347
348         class StateListener implements ControllerListener {
349
350             public void controllerUpdate(ControllerEvent ce) {
351
352                 // If there was an error during configure or
353                 // realize, the processor will be closed
354                 if (ce instanceof ControllerClosedEvent)
355                     setFailed();
356
357                 // All controller events, send a notification
358                 // to the waiting thread in waitForState method
359                 if (ce instanceof ControllerEvent) {
360                     synchronized (getStateLock()) {
361                         getStateLock().notifyAll();
362                     }
363                 }
364             }
365         }
366 }
```

Simulator/Client/ClientRTPSocketAdapter.java

The content of this file is the same as the content of the prototype client file `Client/ClientRTPSocketAdapter.java` printed in section A.1.1.

Simulator/Client/Plugin/H263VideoSignerBasic.java

```
1  package Client.Plugin;
2
3  import java.io.*;
4  import java.util.*;
5  import javax.media.*;
6  import javax.media.format.*;
7  import javax.media.format.AudioFormat;
8  import java.security.*;
9
10 public class H263VideoSignerBasic implements Codec {
11
12     private static String CodecName="H263VideoSignerBasic";
13
14     /** chosen input Format **/
15     protected VideoFormat inputFormat;
16
17     /** chosen output Format **/
18     protected VideoFormat outputFormat;
19
20     /** supported input Formats **/
21     protected Format[] supportedInputFormats=new Format[0];
22
23     /** supported output Formats **/
24     protected Format[] supportedOutputFormats=new Format[0];
25
26     // Buffer number
27     int buffNbr = -1;
28
29     // JCA variables
30     PrivateKey privKey;
31     Signature dsa;
32
33     /** initialize the plugin **/
34     public H263VideoSignerBasic() {
35
36         // Set the input and output formats of this plugin
37         supportedInputFormats = new Format[] {
38             new VideoFormat(VideoFormat.H263_RTP)
39         };
40         supportedOutputFormats = new Format[] {
41             new VideoFormat(VideoFormat.H263_RTP)
42         };
43
44         // Read the private key
45         File f = new File("./privKey.data");
46
47         FileInputStream fis = null;
48
49         try {
50             fis = new FileInputStream(f);
51         }
52         catch(FileNotFoundException e) {
53             error("File not found exception");
54         }
55
56         try {
57             ObjectInput ois = new ObjectInputStream(fis);
58
59             try {
60                 privKey = (PrivateKey) ois.readObject();
61             }
62             catch(ClassNotFoundException e) {
63                 error("Class not found exception");
64             }
65
66             ois.close();
67
68             info("Private key file read");
69         }
70         catch(IOException e) {
71             error("I/O exception");
```

```
72          }
73
74          // Signature initialization
75          try {
76              dsa = Signature.getInstance("SHA1withDSA");
77              dsa.initSign(privKey);
78          }
79          catch(NoSuchAlgorithmException e) {
80              error("No such algorithm exception");
81          }
82          catch(InvalidKeyException e) {
83              error("Invalid key exception");
84          }
85      }
86
87      /** get the resources needed by this plugin **/
88      public void open() throws ResourceUnavailableException {
89      }
90
91      /** free the resources allocated by this plugin **/
92      public void close() {
93      }
94
95      /** reset the plugin **/
96      public void reset() {
97      }
98
99      /** no controls for this simple plugin **/
100     public Object[] getControls() {
101         return (Object[]) new Control[0];
102     }
103
104     /** Return the control based on a control type for the plugin **/
105     public Object getControl(String controlType) {
106         try {
107             Class cls = Class.forName(controlType);
108             Object cs[] = getControls();
109             for (int i = 0; i < cs.length; i++) {
110                 if (cls.isInstance(cs[i]))
111                     return cs[i];
112             }
113             return null;
114         } catch (Exception e) { // no such controlType or such control
115             return null;
116         }
117     }
118
119     /************** format methods *************/
120     /** set the input format **/
121     public Format setInputFormat(Format input) {
122         // the following code assumes valid Format
123         inputFormat = (VideoFormat)input;
124         return (Format)inputFormat;
125     }
126
127     /** set the output format **/
128     public Format setOutputFormat(Format output) {
129         // the following code assumes valid Format
130         outputFormat = (VideoFormat)output;
131         return (Format)outputFormat;
132     }
133
134     /** get the input format **/
135     protected Format getInputFormat() {
136         return inputFormat;
137     }
138
139     /** get the output format **/
140     protected Format getOutputFormat() {
141         return outputFormat;
142     }
143
144     /** supported input formats **/
145     public Format [] getSupportedInputFormats() {
146         return supportedInputFormats;
147     }
148
149     /** output Formats for the selected input format **/
150     public Format [] getSupportedOutputFormats(Format in) {
151         if (in == null)
152             return supportedOutputFormats;
153         else {
154             Format outs[] = new Format[1];
```

```
155             outs[0] = in;
156             return outs;
157         }
158     }
159 
160     /** return plugin name **/
161     public String getName() {
162         return CodecName;
163     }
164 
165     /** do the processing **/
166     public int process(Buffer inputBuffer, Buffer outputBuffer){
167 
168         // Compute the buffer number
169         buffNbr++;
170 
171         if(buffNbr < 0)
172             buffNbr = 0;
173 
174         // Get the input buffer info
175         int inLength = inputBuffer.getLength();
176         int inOffset = inputBuffer.getOffset();
177         byte[] inData =
178             getValidData((byte[]) inputBuffer.getData(),inOffset,inLength);
179 
180         // Print some info
181         info("*******************************");
182         info("Buffer #" + buffNbr);
183         info("Input buffer length: " + inLength);
184         info("Format/encoding : " + inputBuffer.getFormat().getEncoding());
185 
186         Vector buffTmp = new Vector();
187         byte[] outData = null;
188 
189         // Insert inData
190         buffTmp.add(inData);
191 
192         // Insert buffNrb
193         buffTmp.add(int2ByteArray(buffNbr));
194 
195         // Compute and insert signature
196         byte[] sigTmp = computeSignature(vect2ByteArray(buffTmp));
197         buffTmp.add(sigTmp);
198 
199         // Last byte of the output buffer the signature size
200         byte[] lastByte = new byte[] {(byte) (sigTmp.length & 0xff)};
201         buffTmp.add(lastByte);
202 
203         outData = vect2ByteArray(buffTmp);
204 
205         // Set the output buffer content and properties
206         outputBuffer.setData(outData);
207         outputBuffer.setFormat(inputBuffer.getFormat());
208         outputBuffer.setLength(outData.length);
209         outputBuffer.setOffset(0);
210 
211         return BUFFER_PROCESSED_OK;
212     }
213 
214     // Return the signature associated to data
215     private byte[] computeSignature(byte[] data) {
216         byte[] sig = null;
217 
218         try {
219             dsa.update(data);
220             sig = dsa.sign();
221         }
222         catch(SignatureException e) {
223             error("Signature Exception");
224         }
225         return sig;
226     }
227 
228     // Utils
229 
230     // Convert an integer to a byte array
231     private byte[] int2ByteArray(int val) {
232         byte[] tmp = new byte[4];
233         tmp[0] = (byte) val;
234         tmp[1] = (byte) (val >> 8);
235         tmp[2] = (byte) (val >> 16);
236         tmp[3] = (byte) (val >> 24);
237         return tmp;
```

```
238     }
239
240     // Convert a vector of byte arrays to a single byte array
241     // by appending the vector components
242     private byte[] vect2ByteArray(Vector vectOutData) {
243
244         byte[] outData;
245         int vectLength = vectOutData.size();
246
247         int i,length;
248
249         for(i=0,length=0;i<vectLength;i++)
250             length = length + ((byte[]) (vectOutData.get(i))).length;
251
252         outData = new byte[length];
253
254         if(outData == null)
255             error("outData buffer not allocated");
256
257         int j,outDataActualLength;
258         byte[] tmp;
259
260         for(i=0,outDataActualLength=0;i<vectLength;i++) {
261             tmp = (byte[]) vectOutData.get(i);
262             for(j=0;j<tmp.length;j++)
263                 outData[outDataActualLength + j] = tmp[j];
264             outDataActualLength = outDataActualLength + tmp.length;
265         }
266         return outData;
267     }
268
269     // Extract the valid video data from the input buffer
270     private byte[] getValidData(byte[] data,int offset,int length) {
271
272         byte[] tmp = new byte[length];
273         int i;
274
275         for(i=0;i<length;i++)
276             tmp[i] = data[offset + i];
277
278         return tmp;
279     }
280
281     private void info(String msg) {
282         System.out.println("[INFO] H263 Video Signer Plugin: " + msg);
283     }
284
285     private void error(String msg) {
286         System.err.println("[ERROR] H263 Video Signer Plugin: " + msg);
287         System.exit(1);
288     }
289 }
```

Simulator/Client/Plugin/H263VideoSignerSimple.java

```
1  package Client.Plugin;
2
3  import java.io.*;
4  import java.util.*;
5  import javax.media.*;
6  import javax.media.format.*;
7  import javax.media.format.AudioFormat;
8  import java.security.*;
9
10 public class H263VideoSignerSimple implements Codec {
11
12     private static String CodecName="H263VideoSignerSimple";
13
14     /** chosen input Format **/
15     protected VideoFormat inputFormat;
16
17     /** chosen output Format **/
18     protected VideoFormat outputFormat;
19
20     /** supported input Formats **/
21     protected Format[] supportedInputFormats=new Format[0];
22
23     /** supported output Formats **/
24     protected Format[] supportedOutputFormats=new Format[0];
25
26     // Buffer number
27     int buffNbr = -1;
28
29     // Hash size
30     int hashSize = 16;
31
32     // Count the number of buffers left before signature embedding
33     int count;
34
35     // Temporary hash
36     byte[] hashTmp = new byte[hashSize];
37
38     // The hasher
39     MessageDigest digester = null;
40
41     // JCA variables
42     PrivateKey privKey;
43     Signature dsa;
44
45     // (Maximum) Chain length
46     public int chainLength = 16;
47
48     // If set to 'false' fixed length chains are used
49     // Otherwise random length chains are used
50     // with chainLength for maximum length
51     public boolean randomChainLength = true;
52     Random randomEngine = null;
53
54     /** initialize the plugin **/
55     public H263VideoSignerSimple() {
56
57         // Set the input and output formats of this plugin
58         supportedInputFormats = new Format[] {
59             new VideoFormat(VideoFormat.H263_RTP)
60         };
61         supportedOutputFormats = new Format[] {
62             new VideoFormat(VideoFormat.H263_RTP)
63         };
64
65         if(randomChainLength)
66             randomEngine = new Random();
67         count = getCountValue();
68
69         // Initialize the hasher
70         try {
71             digester = MessageDigest.getInstance("MD5");
72         }
73         catch(NoSuchAlgorithmException e) {
74             error("No such algorithm exception");
75         }
76
77         // Read the private key
78         File f = new File("./privKey.data");
79
80         FileInputStream fis = null;
```

```
81
82          try {
83              fis = new FileInputStream(f);
84          }
85          catch(FileNotFoundException e) {
86              error("File not found exception");
87          }
88
89          try {
90              ObjectInput ois = new ObjectInputStream(fis);
91
92              try {
93                  privKey = (PrivateKey) ois.readObject();
94              }
95              catch(ClassNotFoundException e) {
96                  error("Class not found exception");
97              }
98
99              ois.close();
100
101             info("Private key file read");
102         }
103         catch(IOException e) {
104             error("I/O exception");
105         }
106
107         // Signature initialization
108         try {
109             dsa = Signature.getInstance("SHA1withDSA");
110             dsa.initSign(privKey);
111         }
112         catch(NoSuchAlgorithmException e) {
113             error("No such algorithm exception");
114         }
115         catch(InvalidKeyException e) {
116             error("Invalid key exception");
117         }
118     }
119
120     /** get the resources needed by this plugin **/
121     public void open() throws ResourceUnavailableException {
122     }
123
124     /** free the resources allocated by this plugin **/
125     public void close() {
126     }
127
128     /** reset the plugin **/
129     public void reset() {
130     }
131
132     /** no controls for this simple plugin **/
133     public Object[] getControls() {
134         return (Object[]) new Control[0];
135     }
136
137     /** Return the control based on a control type for the plugin **/
138     public Object getControl(String controlType) {
139         try {
140             Class cls = Class.forName(controlType);
141             Object cs[] = getControls();
142             for (int i = 0; i < cs.length; i++) {
143                 if (cls.isInstance(cs[i]))
144                     return cs[i];
145             }
146             return null;
147         } catch (Exception e) { // no such controlType or such control
148             return null;
149         }
150     }
151
152     /************** format methods *************/
153     /** set the input format **/
154     public Format setInputFormat(Format input) {
155         // the following code assumes valid Format
156         inputFormat = (VideoFormat)input;
157         return (Format)inputFormat;
158     }
159
160     /** set the output format **/
161     public Format setOutputFormat(Format output) {
162         // the following code assumes valid Format
163         outputFormat = (VideoFormat)output;
```

```
164         return (Format)outputFormat;
165     }
166 
167     /** get the input format **/
168     protected Format getInputFormat() {
169         return inputFormat;
170     }
171 
172     /** get the output format **/
173     protected Format getOutputFormat() {
174         return outputFormat;
175     }
176 
177     /** supported input formats **/
178     public Format [] getSupportedInputFormats() {
179         return supportedInputFormats;
180     }
181 
182     /** output Formats for the selected input format **/
183     public Format [] getSupportedOutputFormats(Format in) {
184         if (in == null)
185             return supportedOutputFormats;
186         else {
187             Format outs[] = new Format[1];
188             outs[0] = in;
189             return outs;
190         }
191     }
192 
193     /** return plugin name **/
194     public String getName() {
195         return CodecName;
196     }
197 
198     /** do the processing **/
199     public int process(Buffer inputBuffer, Buffer outputBuffer){
200 
201         // Compute the buffer number
202         buffNbr++;
203 
204         if(buffNbr < 0)
205             buffNbr = 0;
206 
207         // count-1 buffers left before signature embedding
208         count--;
209 
210         // Get the input buffer info
211         int inLength = inputBuffer.getLength();
212         int inOffset = inputBuffer.getOffset();
213         byte[] inData =
214             getValidData((byte[]) inputBuffer.getData(),inOffset,inLength);
215 
216         // Print some info
217         info("*******************************");
218         info("Buffer #" + buffNbr);
219         info("Input buffer length: " + inLength);
220         info("Format/encoding : " + inputBuffer.getFormat().getEncoding());
221         info("Buffer left before signature: " + count);
222 
223         Vector buffTmp = new Vector();
224         byte[] outData = null;
225 
226         // Insert inData
227         buffTmp.add(inData);
228 
229         // Insert buffNrb
230         buffTmp.add(int2ByteArray(buffNbr));
231 
232         // Insert hash
233         buffTmp.add(hashTmp);
234 
235         if(count == 0){
236             // Compute and insert signature
237             byte[] sigTmp = computeSignature(vect2ByteArray(buffTmp));
238             buffTmp.add(sigTmp);
239 
240             // Last byte of the output buffer the signature size + 0x80
241             // Then a value of at least 0x80 indicates the presence of a
242             // signature (maximum signature size is 0x2F)
243             byte[] lastByte = new byte[] {(byte) ((sigTmp.length & 0xff) | 0x80)};
244             buffTmp.add(lastByte);
245 
246             count = getCountValue();
```

```
247             info("Signature added");
248         }
249         else {
250             // Compute and add hash to hashQueue
251             byte[] lastByte = {(byte) 0x00};
252             buffTmp.add(lastByte);
253 
254             hashTmp = computeHash(vect2ByteArray(buffTmp));
255         }
256 
257         outData = vect2ByteArray(buffTmp);
258 
259         // Set the output buffer content and properties
260         outputBuffer.setData(outData);
261         outputBuffer.setFormat(inputBuffer.getFormat());
262         outputBuffer.setLength(outData.length);
263         outputBuffer.setOffset(0);
264 
265         return BUFFER_PROCESSED_OK;
266     }
267 
268     // Return a random value greater than 0 and smaller than chainLength + 1 if
269     // randomChainLength is true
270     // Otherwise, return chainLength
271     private int getCountValue() {
272         if(!randomChainLength)
273             return chainLength;
274         else
275             // Return value between 1 and chainLength
276             return ((int) ((float) (chainLength - 1)
277                            * randomEngine.nextFloat())) + 1;
278     }
279     // Return the hash value of data
280     private byte[] computeHash(byte[] data) {
281         return digester.digest(data);
282     }
283 
284     // Return the signature associated to data
285     private byte[] computeSignature(byte[] data) {
286         byte[] sig = null;
287 
288         try {
289             dsa.update(data);
290             sig = dsa.sign();
291         }
292         catch(SignatureException e) {
293             error("Signature Exception");
294         }
295         return sig;
296     }
297 
298     // Utils
299 
300     // Convert an integer to a byte array
301     private byte[] int2ByteArray(int val) {
302         byte[] tmp = new byte[4];
303         tmp[0] = (byte) val;
304         tmp[1] = (byte) (val >> 8);
305         tmp[2] = (byte) (val >> 16);
306         tmp[3] = (byte) (val >> 24);
307         return tmp;
308     }
309 
310     // Convert a vector of byte arrays to a single byte array
311     // by appending the vector components
312     private byte[] vect2ByteArray(Vector vectOutData) {
313 
314         byte[] outData;
315         int vectLength = vectOutData.size();
316 
317         int i,length;
318 
319         for(i=0,length=0;i<vectLength;i++)
320             length = length + ((byte[]) (vectOutData.get(i))).length;
321 
322         outData = new byte[length];
323 
324         if(outData == null)
325             error("outData buffer not allocated");
326 
327         int j,outDataActualLength;
328         byte[] tmp;
329 
```

```
330             for(i=0,outDataActualLength=0;i<vectLength;i++) {
331                 tmp = (byte[]) vectOutData.get(i);
332                 for(j=0;j<tmp.length;j++)
333                     outData[outDataActualLength + j] = tmp[j];
334                 outDataActualLength = outDataActualLength + tmp.length;
335             }
336             return outData;
337         }
338 
339         // Extract the valid video data from the input buffer
340         private byte[] getValidData(byte[] data,int offset,int length) {
341 
342             byte[] tmp = new byte[length];
343             int i;
344 
345             for(i=0;i<length;i++)
346                 tmp[i] = data[offset + i];
347 
348             return tmp;
349         }
350 
351         private void info(String msg) {
352             System.out.println("[INFO] H263 Video Signer Plugin: " + msg);
353         }
354 
355         private void error(String msg) {
356             System.err.println("[ERROR] H263 Video Signer Plugin: " + msg);
357             System.exit(1);
358         }
359 }
```

Simulator/Client/Plugin/H263VideoSignerEMSS.java

```
1  package Client.Plugin;
2
3  import java.io.*;
4  import java.util.*;
5  import javax.media.*;
6  import javax.media.format.*;
7  import javax.media.format.AudioFormat;
8  import java.security.*;
9
10 public class H263VideoSignerEMSS implements Codec {
11
12     private static String CodecName="H263VideoSignerEMSS";
13
14     /** chosen input Format **/
15     protected VideoFormat inputFormat;
16
17     /** chosen output Format **/
18     protected VideoFormat outputFormat;
19
20     /** supported input Formats **/
21     protected Format[] supportedInputFormats=new Format[0];
22
23     /** supported output Formats **/
24     protected Format[] supportedOutputFormats=new Format[0];
25
26     // Buffer number
27     int buffNbr = -1;
28
29     // Hash value
30     int hashSize = 16;
31
32     // Current hash distribution scheme
33     Scheme currentScheme;
34
35     // Hash distribution scheme parameters
36     int schemeLength = 4; // Such that hashSize % schemeLength = 0
37     int schemeRange = 16;
38
39     // Signature frenquency: one signature every sigFreq buffer(s)
40     int sigFreq = 10;
41
42     // Hash queue
43     Vector hashQueue = new Vector();
44
45     // The hasher
46     MessageDigest digester = null;
47
48     // JCA variables
49     PrivateKey privKey;
50     Signature dsa;
51
52     /** initialize the plugin **/
53     public H263VideoSignerEMSS() {
54
55         // Set the input and output formats of this plugin
56         supportedInputFormats = new Format[] {
57             new VideoFormat(VideoFormat.H263_RTP)
58         };
59         supportedOutputFormats = new Format[] {
60             new VideoFormat(VideoFormat.H263_RTP)
61         };
62
63         // Initialize the hasher
64         try {
65             digester = MessageDigest.getInstance("MD5");
66         }
67         catch(NoSuchAlgorithmException e) {
68             error("No such algorithm exception");
69         }
70
71         // Read the private key
72         File f = new File("./privKey.data");
73
74         FileInputStream fis = null;
75
76         try {
77             fis = new FileInputStream(f);
78         }
79         catch(FileNotFoundException e) {
80             error("File not found exception");
```

```
81          }
82
83          try {
84              ObjectInput ois = new ObjectInputStream(fis);
85
86              try {
87                  privKey = (PrivateKey) ois.readObject();
88              }
89              catch(ClassNotFoundException e) {
90                  error("Class not found exception");
91              }
92
93              ois.close();
94
95              info("Private key file read");
96          }
97          catch(IOException e) {
98              error("I/O exception");
99          }
100
101         // Signature initialization
102         try {
103             dsa = Signature.getInstance("SHA1withDSA");
104             dsa.initSign(privKey);
105         }
106         catch(NoSuchAlgorithmException e) {
107             error("No such algorithm exception");
108         }
109         catch(InvalidKeyException e) {
110             error("Invalid key exception");
111         }
112     }
113
114     /** get the resources needed by this plugin **/
115     public void open() throws ResourceUnavailableException {
116     }
117
118     /** free the resources allocated by this plugin **/
119     public void close() {
120     }
121
122     /** reset the plugin **/
123     public void reset() {
124     }
125
126     /** no controls for this simple plugin **/
127     public Object[] getControls() {
128         return (Object[]) new Control[0];
129     }
130
131     /** Return the control based on a control type for the plugin **/
132     public Object getControl(String controlType) {
133         try {
134             Class cls = Class.forName(controlType);
135             Object cs[] = getControls();
136             for (int i = 0; i < cs.length; i++) {
137                 if (cls.isInstance(cs[i]))
138                     return cs[i];
139             }
140             return null;
141         } catch (Exception e) { // no such controlType or such control
142             return null;
143         }
144     }
145
146     /************** format methods *************/
147     /** set the input format **/
148     public Format setInputFormat(Format input) {
149         // the following code assumes valid Format
150         inputFormat = (VideoFormat)input;
151         return (Format)inputFormat;
152     }
153
154     /** set the output format **/
155     public Format setOutputFormat(Format output) {
156         // the following code assumes valid Format
157         outputFormat = (VideoFormat)output;
158         return (Format)outputFormat;
159     }
160
161     /** get the input format **/
162     protected Format getInputFormat() {
163         return inputFormat;
```

```
164     }
165
166     /** get the output format **/
167     protected Format getOutputFormat() {
168         return outputFormat;
169     }
170
171     /** supported input formats **/
172     public Format [] getSupportedInputFormats() {
173         return supportedInputFormats;
174     }
175
176     /** output Formats for the selected input format **/
177     public Format [] getSupportedOutputFormats(Format in) {
178         if (in == null)
179             return supportedOutputFormats;
180         else {
181             Format outs[] = new Format[1];
182             outs[0] = in;
183             return outs;
184         }
185     }
186
187     /** return plugin name **/
188     public String getName() {
189         return CodecName;
190     }
191
192     /** do the processing **/
193     public int process(Buffer inputBuffer, Buffer outputBuffer){
194
195         // Compute the buffer number
196         buffNbr++;
197
198         if(buffNbr < 0)
199             buffNbr = 0;
200
201         // Get the input buffer info
202         int inLength = inputBuffer.getLength();
203         int inOffset = inputBuffer.getOffset();
204         byte[] inData =
205             getValidData((byte[]) inputBuffer.getData(),inOffset,inLength);
206
207         // Print some info
208         info("*******************************");
209         info("Buffer #" + buffNbr);
210         info("Input buffer length: " + inLength);
211         info("Format/encoding : " + inputBuffer.getFormat().getEncoding());
212
213         // Compute the current hash distribution scheme
214         currentScheme = new Scheme(schemeLength,schemeRange);
215
216         // Determine which hashes have to be appended
217         Vector hashList = getHashList();
218
219         // Compute the count byte
220         int hashListSize = hashList.size();
221
222         info("Number of hash: " + hashListSize);
223
224         info("hashQueueSize: " + hashQueue.size());
225
226         byte[] hashCount = new byte[] {(byte) (hashListSize & 0xff)};
227
228         // Compute the output data
229         Vector dataVect = new Vector();
230
231         dataVect.add(inData);
232
233         int i;
234         for(i=0;i<hashListSize;i++) {
235             byte[] tmp = (byte[]) hashList.get(i);
236             dataVect.add(tmp);
237         }
238
239         dataVect.add(hashCount);
240         dataVect.add(int2ByteArray(buffNbr));
241
242         byte[] outData = vect2ByteArray(dataVect);
243
244         // Compute and add the hash to the hash queue
245         hashQueue.add(generateHash(outData));
246
```

```
247         // Determine if a signature has to be appended
248         if(buffNbr % sigFreq == 0) {
249 
250             info("Signed buffer");
251 
252             byte[] sig = null;
253 
254             try {
255                 dsa.update(outData);
256                 sig = dsa.sign();
257             }
258             catch(SignatureException e) {
259                 error("Signature Exception");
260             }
261 
262             // Insertion of the signature
263             byte[] tmpData = new byte[outData.length + sig.length + 1];
264             // Last byte of the output buffer the signature size + 0x80
265             // Then a value of at least 0x80 indicates the presence of a
266             // signature (maximum signature size is 0x2F)
267             byte[] sigSize = new byte[] {(byte) ((sig.length & 0xff) | 0x80)};
268 
269             for(i=0;i<outData.length-5;i++)
270                 tmpData[i] = outData[i];
271 
272             tmpData[i] = hashCount[0];
273 
274             i++;
275 
276             int j;
277             for(j=0;j<sig.length;j++)
278                 tmpData[i+j] = sig[j];
279 
280             i=i+j;
281 
282             tmpData[i] = sigSize[0];
283 
284             i++;
285 
286             for(j=0;j<4;j++)
287                 tmpData[i+j] = int2ByteArray(buffNbr)[j];
288 
289             outData = tmpData;
290         }
291 
292         // Clean the hash queue
293         if(hashQueue.size() > schemeRange)
294             hashQueue.removeElementAt(0);
295 
296         // Set the output buffer content and properties
297         outputBuffer.setData(outData);
298         outputBuffer.setFormat(inputBuffer.getFormat());
299         outputBuffer.setLength(outData.length);
300         outputBuffer.setOffset(0);
301 
302         return BUFFER_PROCESSED_OK;
303     }
304 
305     // Determine which hashes have to be appended
306     private Vector getHashList() {
307 
308         Vector hashList = new Vector();
309 
310         int hashQueueSize = hashQueue.size();
311 
312         int i,j;
313         HashQueueItem hQItemTmp;
314 
315         for(i=0;i<hashQueueSize;i++) {
316 
317             hQItemTmp = (HashQueueItem) hashQueue.get(i);
318 
319             for(j=0;j<hQItemTmp.destList.size();j++)
320                 if(buffNbr == ((Integer) hQItemTmp.destList.get(j)).intValue()) {
321                     // Add hashPart number (lastPart + 1) % schemeLength
322                     hQItemTmp.lastPart++;
323                     hashList.add((byte[]) hQItemTmp.hashParts.get(hQItemTmp.lastPart ↩
                            % schemeLength));
324                     hQItemTmp.destList.removeElementAt(j);
325                     j--;
326                 }
327         }
328         return hashList;
```

```
329     }
330
331     // Compute the hash of a buffer
332     private HashQueueItem generateHash(byte[] data) {
333
334         HashQueueItem hQItemTmp = new HashQueueItem();
335
336         // Set the source buffer number
337         hQItemTmp.srcBuffer = buffNbr;
338
339         // Set the destination buffer number list
340         int i,j;
341         for(i=0;i<schemeLength;i++)
342             hQItemTmp.destList.add(
343                         new Integer(buffNbr + currentScheme.schemeData[i])
344                         );
345
346         byte[] hash = digester.digest(data);
347
348         int partLength = (hashSize / schemeLength) + 5;
349         byte[] partTmp;
350
351         byte[] buffNbrByte = int2ByteArray(buffNbr);
352
353         byte partNbr = 0;
354         for(i=0;i<hashSize;i=i+partLength-5) {
355
356             partTmp = new byte[partLength];
357
358             for(j=0;j<partLength-5;j++)
359                 partTmp[j] = hash[i+j];
360
361             partTmp[j] = partNbr;
362             partNbr++;
363
364             for(j=0;j<4;j++)
365                 partTmp[partLength - 4 + j] = buffNbrByte[j];
366
367             hQItemTmp.hashParts.add(partTmp);
368         }
369
370         hQItemTmp.lastPart = -1;
371
372         return hQItemTmp;
373     }
374
375     // Extract the valid video data from the input buffer
376     private byte[] getValidData(byte[] data,int offset,int length) {
377
378         byte[] tmp = new byte[length];
379         int i;
380
381         for(i=0;i<length;i++)
382             tmp[i] = data[offset + i];
383
384         return tmp;
385     }
386
387     // Utils
388
389     // Convert an integer to a byte array
390     private byte[] int2ByteArray(int val) {
391         byte[] tmp = new byte[4];
392         tmp[0] = (byte) val;
393         tmp[1] = (byte) (val >> 8);
394         tmp[2] = (byte) (val >> 16);
395         tmp[3] = (byte) (val >> 24);
396         return tmp;
397     }
398
399     // Convert a vector of byte arrays to a single byte array
400     // by appending the vector components
401     private byte[] vect2ByteArray(Vector vectOutData) {
402
403         byte[] outData;
404         int vectLength = vectOutData.size();
405
406         int i,length;
407
408         for(i=0,length=0;i<vectLength;i++)
409             length = length + ((byte[]) (vectOutData.get(i))).length;
410
411         outData = new byte[length];
```

```
412
413             if(outData == null)
414                 error("outData buffer not allocated");
415
416             int j,outDataActualLength;
417             byte[] tmp;
418
419             for(i=0,outDataActualLength=0;i<vectLength;i++) {
420                 tmp = (byte[]) vectOutData.get(i);
421                 for(j=0;j<tmp.length;j++)
422                     outData[outDataActualLength + j] = tmp[j];
423                 outDataActualLength = outDataActualLength + tmp.length;
424             }
425             return outData;
426         }
427
428         private void info(String msg) {
429             System.out.println("[INFO] H263 Video Signer Plugin: " + msg);
430         }
431
432         private void error(String msg) {
433             System.err.println("[ERROR] H263 Video Signer Plugin: " + msg);
434             System.exit(1);
435         }
436
437         // Internal classes
438
439         // Class defining the hash distribution scheme
440         class Scheme {
441
442             int[] schemeData;
443
444             public Scheme(int length, int range) {
445
446                 TreeSet data;
447
448                 data = new TreeSet();
449                 Random randomEngine = new Random();
450
451                 int i,tmp;
452                 Integer val;
453
454                 for(i=0;i<length;i++) {
455                     val = new Integer((int) ((float) range * randomEngine.nextFloat()));
456                     tmp = val.intValue();
457                     if(tmp == 0)
458                         val = new Integer(tmp + 1);
459                     if(data.contains(val))
460                         i--;
461                     else
462                         data.add(val);
463                 }
464
465                 Iterator iterator = data.iterator();
466                 schemeData = new int[length];
467
468                 for(i=0;i<length;i++)
469                     schemeData[i] = ((Integer) (iterator.next())).intValue();
470             }
471
472             public void printScheme() {
473
474                 int i;
475
476                 for(i=0;i<schemeData.length;i++)
477                     System.out.println("schemeData[" + i + "] = " + schemeData[i]);
478             }
479         }
480
481         // Class defining the items of the hash queue
482         class HashQueueItem {
483
484             // Source buffer number
485             int srcBuffer;
486
487             // Source
488             int lastPart;
489
490             // Hash part list
491             Vector hashParts;
492
493             // Destination buffer number list
494             Vector destList;
```

```
495
496             public HashQueueItem() {
497                 hashParts = new Vector();
498                 destList = new Vector();
499             }
500         }
501 }
```

Simulator/Client/Plugin/H263VideoSignerStar.java

```
1  package Client.Plugin;
2
3  import java.io.*;
4  import java.util.*;
5  import javax.media.*;
6  import javax.media.format.*;
7  import javax.media.format.AudioFormat;
8  import java.security.*;
9
10 public class H263VideoSignerStar implements Codec {
11
12     private static String CodecName="H263VideoSignerStar";
13
14     /** chosen input Format **/
15     protected VideoFormat inputFormat;
16
17     /** chosen output Format **/
18     protected VideoFormat outputFormat;
19
20     /** supported input Formats **/
21     protected Format[] supportedInputFormats=new Format[0];
22
23     /** supported output Formats **/
24     protected Format[] supportedOutputFormats=new Format[0];
25
26     // Buffer number
27     int buffNbr = -1;
28
29     // Block number
30     int blockNbr = -1;
31
32     // Input buffer queue
33     Vector inputBufferQueue = new Vector();
34
35     // Output buffer queue
36     Vector outputBufferQueue = new Vector();
37
38     // Number of buffer per block
39     public int blockSize = 8;
40
41     // Hash size
42     int hashSize = 16;
43
44     // The hasher
45     MessageDigest digester = null;
46
47     // JCA variables
48     PrivateKey privKey;
49     Signature dsa;
50
51     /** initialize the plugin **/
52     public H263VideoSignerStar() {
53
54         // Set the input and output formats of this plugin
55         supportedInputFormats = new Format[] {
56             new VideoFormat(VideoFormat.H263_RTP)
57         };
58         supportedOutputFormats = new Format[] {
59             new VideoFormat(VideoFormat.H263_RTP)
60         };
61
62         // Initialize the hasher
63         try {
64             digester = MessageDigest.getInstance("MD5");
65         }
66         catch(NoSuchAlgorithmException e) {
67             error("No such algorithm exception");
68         }
69
70         // Read the private key
71         File f = new File("./privKey.data");
72
73         FileInputStream fis = null;
74
75         try {
76             fis = new FileInputStream(f);
77         }
78         catch(FileNotFoundException e) {
79             error("File not found exception");
80         }
```

```
81
82          try {
83              ObjectInput ois = new ObjectInputStream(fis);
84
85              try {
86                  privKey = (PrivateKey) ois.readObject();
87              }
88              catch(ClassNotFoundException e) {
89                  error("Class not found exception");
90              }
91
92              ois.close();
93
94              info("Private key file read");
95          }
96          catch(IOException e) {
97              error("I/O exception");
98          }
99
100         // Signature initialization
101         try {
102             dsa = Signature.getInstance("SHA1withDSA");
103             dsa.initSign(privKey);
104         }
105         catch(NoSuchAlgorithmException e) {
106             error("No such algorithm exception");
107         }
108         catch(InvalidKeyException e) {
109             error("Invalid key exception");
110         }
111     }
112
113     /** get the resources needed by this plugin **/
114     public void open() throws ResourceUnavailableException {
115     }
116
117     /** free the resources allocated by this plugin **/
118     public void close() {
119     }
120
121     /** reset the plugin **/
122     public void reset() {
123     }
124
125     /** no controls for this simple plugin **/
126     public Object[] getControls() {
127         return (Object[]) new Control[0];
128     }
129
130     /** Return the control based on a control type for the plugin **/
131     public Object getControl(String controlType) {
132         try {
133             Class cls = Class.forName(controlType);
134             Object cs[] = getControls();
135             for (int i = 0; i < cs.length; i++) {
136                 if (cls.isInstance(cs[i]))
137                     return cs[i];
138             }
139             return null;
140         } catch (Exception e) { // no such controlType or such control
141             return null;
142         }
143     }
144
145     /************** format methods *************/
146     /** set the input format **/
147     public Format setInputFormat(Format input) {
148         // the following code assumes valid Format
149         inputFormat = (VideoFormat)input;
150         return (Format)inputFormat;
151     }
152
153     /** set the output format **/
154     public Format setOutputFormat(Format output) {
155         // the following code assumes valid Format
156         outputFormat = (VideoFormat)output;
157         return (Format)outputFormat;
158     }
159
160     /** get the input format **/
161     protected Format getInputFormat() {
162         return inputFormat;
163     }
```

```
164
165     /** get the output format **/
166     protected Format getOutputFormat() {
167         return outputFormat;
168     }
169
170     /** supported input formats **/
171     public Format [] getSupportedInputFormats() {
172         return supportedInputFormats;
173     }
174
175     /** output Formats for the selected input format **/
176     public Format [] getSupportedOutputFormats(Format in) {
177         if (in == null)
178             return supportedOutputFormats;
179         else {
180             Format outs[] = new Format[1];
181             outs[0] = in;
182             return outs;
183         }
184     }
185
186     /** return plugin name **/
187     public String getName() {
188         return CodecName;
189     }
190
191     /** do the processing **/
192     public int process(Buffer inputBuffer, Buffer outputBuffer){
193
194         // Compute the buffer number
195         buffNbr++;
196
197         if(buffNbr < 0)
198             buffNbr = 0;
199
200         // Get the input buffer info
201         int inLength = inputBuffer.getLength();
202         int inOffset = inputBuffer.getOffset();
203         byte[] inData =
204             getValidData((byte[]) inputBuffer.getData(),inOffset,inLength);
205
206         // Print some info
207         info("*******************************");
208         info("Buffer #" + buffNbr);
209         info("Input buffer length: " + inLength);
210         info("Format/encoding : " + inputBuffer.getFormat().getEncoding());
211
212         // Add current buffer and its hash to input buffer queue
213         Vector dataVectTmp = new Vector();
214         dataVectTmp.add(inData);
215         dataVectTmp.add(int2ByteArray(buffNbr));
216         byte[] dataTmp = vect2ByteArray(dataVectTmp);
217
218         inputBufferQueue.add(new InputBufferQueueItem(dataTmp,computeHash(dataTmp)));
219
220         info("Input buffer queue size: " + inputBufferQueue.size());
221         info("Output buffer queue size: " + outputBufferQueue.size());
222
223         if(inputBufferQueue.size() == blockSize) {
224             // Generate an authentication star from the input buffer queue
225             // and fill the output buffer queue
226             blockNbr++;
227             processBlock();
228             info("Block #" + blockNbr + " processed");
229         }
230
231         if(outputBufferQueue.size() > 0) {
232             byte[] outData = (byte[]) outputBufferQueue.get(0);
233             outputBufferQueue.remove(0);
234
235             // Set the output buffer content and properties
236             outputBuffer.setData(outData);
237             outputBuffer.setFormat(inputBuffer.getFormat());
238             outputBuffer.setLength(outData.length);
239             outputBuffer.setOffset(0);
240
241             info("Buffer sent");
242             return BUFFER_PROCESSED_OK;
243         }
244         else
245             return OUTPUT_BUFFER_NOT_FILLED;
246     }
```

```
247
248     // Extract the valid video data from the input buffer
249     private byte[] getValidData(byte[] data,int offset,int length) {
250
251         byte[] tmp = new byte[length];
252         int i;
253
254         for(i=0;i<length;i++)
255             tmp[i] = data[offset + i];
256
257         return tmp;
258     }
259
260     // Utils
261
262     // Convert an integer to a byte array
263     private byte[] int2ByteArray(int val) {
264         byte[] tmp = new byte[4];
265         tmp[0] = (byte) val;
266         tmp[1] = (byte) (val >> 8);
267         tmp[2] = (byte) (val >> 16);
268         tmp[3] = (byte) (val >> 24);
269         return tmp;
270     }
271
272     // Convert a vector of byte arrays to a single byte array
273     // by appending the vector components
274     private byte[] vect2ByteArray(Vector vectOutData) {
275
276         byte[] outData;
277         int vectLength = vectOutData.size();
278
279         int i,length;
280
281         for(i=0,length=0;i<vectLength;i++)
282             length = length + ((byte[]) (vectOutData.get(i))).length;
283
284         outData = new byte[length];
285
286         if(outData == null)
287             error("outData buffer not allocated");
288
289         int j,outDataActualLength;
290         byte[] tmp;
291
292         for(i=0,outDataActualLength=0;i<vectLength;i++) {
293             tmp = (byte[]) vectOutData.get(i);
294             for(j=0;j<tmp.length;j++)
295                 outData[outDataActualLength + j] = tmp[j];
296             outDataActualLength = outDataActualLength + tmp.length;
297         }
298         return outData;
299     }
300
301     // Create an authentication star with the buffer present inside the input queue
302     // Create the resulting buffers containing the authentication information and
303     // put them inside the output queue
304     private void processBlock() {
305
306         int i,j;
307         Vector vectTmp = new Vector();
308         byte[] blockSig;
309
310         // Compute block signature
311         for(i=0;i<blockSize;i++)
312             vectTmp.add(((InputBufferQueueItem) inputBufferQueue.get(i)).hash);
313
314         blockSig = computeSignature(computeHash(vect2ByteArray(vectTmp)));
315
316         // Generate output buffers
317         for(i=0;i<blockSize;i++) {
318             vectTmp = new Vector();
319
320             // Add data portion
321             vectTmp.add(((InputBufferQueueItem) inputBufferQueue.get(i)).data);
322
323             // Add hash list portion
324             for(j=0;j<blockSize;j++)
325                 if(i != j)
326                   vectTmp.add(((InputBufferQueueItem) inputBufferQueue.get(j)).hash);
327
328             // Add position in block
329             vectTmp.add(int2ByteArray(i));
```

```
330             // Add block number
331             vectTmp.add(int2ByteArray(blockNbr));
332             // Add block signature
333             vectTmp.add(blockSig);
334             // Add block signature size
335             vectTmp.add(new byte[] {(byte) (blockSig.length & 0xff)});
336 
337             // Add created buffer to output buffer queue
338             outputBufferQueue.add(vect2ByteArray(vectTmp));
339         }
340 
341         // Clean input buffer queue
342         for(i=0;i<blockSize;i++)
343             inputBufferQueue.remove(0);
344     }
345 
346     // Return the hash value of data
347     private byte[] computeHash(byte[] data) {
348         return digester.digest(data);
349     }
350 
351     // Return the signature associated to data
352     private byte[] computeSignature(byte[] data) {
353         byte[] sig = null;
354 
355         try {
356             dsa.update(data);
357             sig = dsa.sign();
358         }
359         catch(SignatureException e) {
360             error("Signature Exception");
361         }
362         return sig;
363     }
364 
365     private void info(String msg) {
366         System.out.println("[INFO] H263 Video Signer Plugin: " + msg);
367     }
368 
369     private void error(String msg) {
370         System.err.println("[ERROR] H263 Video Signer Plugin: " + msg);
371         System.exit(1);
372     }
373 
374     // Class defining the items contained in the input buffer queue
375     class InputBufferQueueItem {
376 
377         // BuffNbr + Buffer
378         byte[] data;
379 
380         // Buffer hash
381         byte[] hash;
382 
383         public InputBufferQueueItem(byte[] data, byte[] hash) {
384             this.data = data;
385             this.hash = hash;
386         }
387     }
388 }
```

Simulator/Client/Plugin/H263VideoSignerTree.java

```
1  package Client.Plugin;
2
3  import java.io.*;
4  import java.util.*;
5  import javax.media.*;
6  import javax.media.format.*;
7  import javax.media.format.AudioFormat;
8  import java.security.*;
9
10 public class H263VideoSignerTree implements Codec {
11
12     private static String CodecName="H263VideoSignerTree";
13
14     /** chosen input Format **/
15     protected VideoFormat inputFormat;
16
17     /** chosen output Format **/
18     protected VideoFormat outputFormat;
19
20     /** supported input Formats **/
21     protected Format[] supportedInputFormats=new Format[0];
22
23     /** supported output Formats **/
24     protected Format[] supportedOutputFormats=new Format[0];
25
26     // Buffer number
27     int buffNbr = -1;
28
29     // Block number
30     int blockNbr = -1;
31
32     // Input buffer queue
33     Vector inputBufferQueue = new Vector();
34
35     // Output buffer queue
36     Vector outputBufferQueue = new Vector();
37
38     // Number of buffer per block
39     public int blockSize = 8;
40
41     // Tree degree such that log(treeDeg)(blockSize) == belongs to the int set
42     public int treeDeg = 2;
43
44     // Hash size
45     int hashSize = 16;
46
47     // The hasher
48     MessageDigest digester = null;
49
50     // JCA variables
51     PrivateKey privKey;
52     Signature dsa;
53
54     Random randomEngine = null;
55
56     /** initialize the plugin **/
57     public H263VideoSignerTree() {
58
59         // Set the input and output formats of this plugin
60         supportedInputFormats = new Format[] {
61             new VideoFormat(VideoFormat.H263_RTP)
62         };
63         supportedOutputFormats = new Format[] {
64             new VideoFormat(VideoFormat.H263_RTP)
65         };
66
67         // Initialize the hasher
68         try {
69             digester = MessageDigest.getInstance("MD5");
70         }
71         catch(NoSuchAlgorithmException e) {
72             error("No such algorithm exception");
73         }
74
75         // Read the private key
76         File f = new File("./privKey.data");
77
78         FileInputStream fis = null;
79
80         try {
```

```
81              fis = new FileInputStream(f);
82          }
83          catch(FileNotFoundException e) {
84              error("File not found exception");
85          }
86
87          try {
88              ObjectInput ois = new ObjectInputStream(fis);
89
90              try {
91                  privKey = (PrivateKey) ois.readObject();
92              }
93              catch(ClassNotFoundException e) {
94                  error("Class not found exception");
95              }
96
97              ois.close();
98
99              info("Private key file read");
100         }
101         catch(IOException e) {
102             error("I/O exception");
103         }
104
105         // Signature initialization
106         try {
107             dsa = Signature.getInstance("SHA1withDSA");
108             dsa.initSign(privKey);
109         }
110         catch(NoSuchAlgorithmException e) {
111             error("No such algorithm exception");
112         }
113         catch(InvalidKeyException e) {
114             error("Invalid key exception");
115         }
116     }
117
118     /** get the resources needed by this plugin **/
119     public void open() throws ResourceUnavailableException {
120     }
121
122     /** free the resources allocated by this plugin **/
123     public void close() {
124     }
125
126     /** reset the plugin **/
127     public void reset() {
128     }
129
130     /** no controls for this simple plugin **/
131     public Object[] getControls() {
132         return (Object[]) new Control[0];
133     }
134
135     /** Return the control based on a control type for the plugin **/
136     public Object getControl(String controlType) {
137         try {
138             Class cls = Class.forName(controlType);
139             Object cs[] = getControls();
140             for (int i = 0; i < cs.length; i++) {
141                 if (cls.isInstance(cs[i]))
142                     return cs[i];
143             }
144             return null;
145         } catch (Exception e) { // no such controlType or such control
146             return null;
147         }
148     }
149
150     /************** format methods *************/
151     /** set the input format **/
152     public Format setInputFormat(Format input) {
153         // the following code assumes valid Format
154         inputFormat = (VideoFormat)input;
155         return (Format)inputFormat;
156     }
157
158     /** set the output format **/
159     public Format setOutputFormat(Format output) {
160         // the following code assumes valid Format
161         outputFormat = (VideoFormat)output;
162         return (Format)outputFormat;
163     }
```

```
164
165     /** get the input format **/
166     protected Format getInputFormat() {
167         return inputFormat;
168     }
169
170     /** get the output format **/
171     protected Format getOutputFormat() {
172         return outputFormat;
173     }
174
175     /** supported input formats **/
176     public Format [] getSupportedInputFormats() {
177         return supportedInputFormats;
178     }
179
180     /** output Formats for the selected input format **/
181     public Format [] getSupportedOutputFormats(Format in) {
182         if (in == null)
183             return supportedOutputFormats;
184         else {
185             Format outs[] = new Format[1];
186             outs[0] = in;
187             return outs;
188         }
189     }
190
191     /** return plugin name **/
192     public String getName() {
193         return CodecName;
194     }
195
196     /** do the processing **/
197     public int process(Buffer inputBuffer, Buffer outputBuffer){
198
199         // Compute the buffer number
200         buffNbr++;
201
202         if(buffNbr < 0)
203             buffNbr = 0;
204
205         // Get the input buffer info
206         int inLength = inputBuffer.getLength();
207         int inOffset = inputBuffer.getOffset();
208         byte[] inData =
209             getValidData((byte[]) inputBuffer.getData(),inOffset,inLength);
210
211         // Print some info
212         info("*****************************");
213         info("Buffer #" + buffNbr);
214         info("Input buffer length: " + inLength);
215         info("Format/encoding : " + inputBuffer.getFormat().getEncoding());
216
217         // Add current buffer and its hash to input buffer queue
218         Vector dataVectTmp = new Vector();
219         dataVectTmp.add(inData);
220         dataVectTmp.add(int2ByteArray(buffNbr));
221         byte[] dataTmp = vect2ByteArray(dataVectTmp);
222
223         inputBufferQueue.add(new InputBufferQueueItem(dataTmp,computeHash(dataTmp)));
224
225         info("Input buffer queue size: " + inputBufferQueue.size());
226         info("Output buffer queue size: " + outputBufferQueue.size());
227
228         if(inputBufferQueue.size() == blockSize) {
229             // Generate an authentication tree from the input buffer queue
230             // and fill the output buffer queue
231             blockNbr++;
232             processBlock();
233             info("Block #" + blockNbr + " processed");
234         }
235
236         if(outputBufferQueue.size() > 0) {
237             byte[] outData = (byte[]) outputBufferQueue.get(0);
238             outputBufferQueue.remove(0);
239
240             // Set the output buffer content and properties
241             outputBuffer.setData(outData);
242             outputBuffer.setFormat(inputBuffer.getFormat());
243             outputBuffer.setLength(outData.length);
244             outputBuffer.setOffset(0);
245
246             info("Buffer sent");
```

```
247             return BUFFER_PROCESSED_OK;
248         }
249         else
250             return OUTPUT_BUFFER_NOT_FILLED;
251     }
252 
253     // Extract the valid video data from the input buffer
254     private byte[] getValidData(byte[] data,int offset,int length) {
255 
256         byte[] tmp = new byte[length];
257         int i;
258 
259         for(i=0;i<length;i++)
260             tmp[i] = data[offset + i];
261 
262         return tmp;
263     }
264 
265     // Utils
266 
267     // Convert an integer to a byte array
268     private byte[] int2ByteArray(int val) {
269         byte[] tmp = new byte[4];
270         tmp[0] = (byte) val;
271         tmp[1] = (byte) (val >> 8);
272         tmp[2] = (byte) (val >> 16);
273         tmp[3] = (byte) (val >> 24);
274         return tmp;
275     }
276 
277     // Convert a vector of byte arrays to a single byte array
278     // by appending the vector components
279     private byte[] vect2ByteArray(Vector vectOutData) {
280 
281         byte[] outData;
282         int vectLength = vectOutData.size();
283 
284         int i,length;
285 
286         for(i=0,length=0;i<vectLength;i++)
287             length = length + ((byte[]) (vectOutData.get(i))).length;
288 
289         outData = new byte[length];
290 
291         if(outData == null)
292             error("outData buffer not allocated");
293 
294         int j,outDataActualLength;
295         byte[] tmp;
296 
297         for(i=0,outDataActualLength=0;i<vectLength;i++) {
298             tmp = (byte[]) vectOutData.get(i);
299             for(j=0;j<tmp.length;j++)
300                 outData[outDataActualLength + j] = tmp[j];
301             outDataActualLength = outDataActualLength + tmp.length;
302         }
303         return outData;
304     }
305 
306     // Create an authentication tree with the buffer present inside the input queue
307     // Create the resulting buffers containing the authentication information and
308     // put them inside the output queue
309     private void processBlock() {
310 
311         int i,j,k;
312         Vector tree = new Vector();
313         Vector lastLevel = new Vector();
314         int lastLevelSize;
315         Vector nextLevel;
316         Vector vectTmp;
317 
318         // Insert the leaves in the tree
319         for(i=0;i<blockSize;i++)
320             lastLevel.add(((InputBufferQueueItem) inputBufferQueue.get(i)).hash);
321         tree.add(lastLevel);
322 
323         // Generate the tree
324         info("Generating tree ...");
325         lastLevelSize = blockSize;
326         while(lastLevelSize > 1) {
327             info("Last level size: " + lastLevelSize);
328             nextLevel = new Vector();
329 
```

```
330             for(i=0;i<lastLevelSize;i+=treeDeg) {
331                 vectTmp = new Vector();
332                 for(j=0;j<treeDeg;j++)
333                     vectTmp.add((byte[]) lastLevel.get(i+j));
334                 nextLevel.add(computeHash(vect2ByteArray(vectTmp)));
335             }
336             lastLevel = nextLevel;
337             tree.add(lastLevel);
338             lastLevelSize = lastLevel.size();
339         }
340         info("Last level size: " + lastLevelSize);
341         info("Tree generated");
342         info("Generating output buffers ...");
343 
344         byte[] blockSig = computeSignature((byte[]) lastLevel.get(0));
345 
346         int posInLevel;
347         int branchNbr;
348         int currentPos;
349 
350         for(i=0;i<blockSize;i++) {
351             vectTmp = new Vector();
352             // Add video data and buffer number
353             vectTmp.add(((InputBufferQueueItem) inputBufferQueue.get(i)).data);
354             // Add hash list
355             posInLevel = i;
356             branchNbr = posInLevel / treeDeg;
357 
358             for(j=0;j<tree.size()-1;j++) {
359                 // Add siblings from level j
360                 for(k=0;k<treeDeg;k++) {
361                     currentPos = (branchNbr * treeDeg) + k;
362                     if(posInLevel != currentPos)
363                         vectTmp.add((byte[]) ((Vector) tree.get(j)).get(currentPos));
364                 }
365 
366                 // Jump to father node
367                 posInLevel = posInLevel / treeDeg;
368                 branchNbr = posInLevel / treeDeg;
369             }
370             // Add buffer position in block
371             vectTmp.add(int2ByteArray(i));
372             // Add block number
373             vectTmp.add(int2ByteArray(blockNbr));
374             // Add signature
375             vectTmp.add(blockSig);
376             // Add signature size
377             vectTmp.add(new byte[] {(byte) (blockSig.length & 0xff)});
378 
379             outputBufferQueue.add(vect2ByteArray(vectTmp));
380         }
381         info("Output buffers generated");
382 
383         // Clean input buffer queue
384         for(i=0;i<blockSize;i++)
385             inputBufferQueue.remove(0);
386     }
387 
388     // Return the hash value of data
389     private byte[] computeHash(byte[] data) {
390         return digester.digest(data);
391     }
392 
393     // Return the signature associated to data
394     private byte[] computeSignature(byte[] data) {
395         byte[] sig = null;
396 
397         try {
398             dsa.update(data);
399             sig = dsa.sign();
400         }
401         catch(SignatureException e) {
402             error("Signature Exception");
403         }
404         return sig;
405     }
406 
407     private void info(String msg) {
408         System.out.println("[INFO] H263 Video Signer Plugin: " + msg);
409     }
410 
411     private void error(String msg) {
412         System.err.println("[ERROR] H263 Video Signer Plugin: " + msg);
```

```
413             System.exit(1);
414         }
415 
416         // Class defining the items contained in the input buffer queue
417         class InputBufferQueueItem {
418 
419             // BuffNbr + Buffer
420             byte[] data;
421 
422             // Buffer hash
423             byte[] hash;
424 
425             public InputBufferQueueItem(byte[] data, byte[] hash) {
426                 this.data = data;
427                 this.hash = hash;
428             }
429         }
430 }
```

A.2.4 Server

Simulator/Server/ServerMain.java

```
1  package Server;
2
3  import java.lang.*;
4  import java.io.*;
5  import java.net.*;
6  import java.util.*;
7  import javax.media.Format;
8  import javax.media.format.VideoFormat;
9  import javax.media.PlugInManager;
10
11 public class ServerMain {
12
13     public static void main(String args[]) {
14
15         ServerRTPReceiver receiver = null;
16
17         if(args.length < 2)
18             prUsage();
19         else {
20
21             String pluginType = "Server.Plugin.H263VideoVerifier" + args[1];
22
23             // Insert the specified H263 Video Verifier plugin into plugin list
24             if(PlugInManager.addPlugIn(pluginType,
25                                        // Plugin input format
26                                        new Format [] {
27                                            new VideoFormat(VideoFormat.H263_RTP)
28                                        },
29                                        // Plugin output format
30                                        new Format [] {
31                                            new VideoFormat(VideoFormat.H263_RTP)
32                                        },
33                                        PlugInManager.CODEC))
34                 info("H263 Video Verifier <" + args[1] + ">: Plugin registered");
35             else
36                 error("H263 Video Verifier: Error while registering "
37                       + args[1] + " plugin");
38
39             // Create and start the RTP receiver
40             try {
41                 receiver =
42                    new ServerRTPReceiver(InetAddress.getLocalHost().getHostAddress(),
43                                          Integer.parseInt(args[0]),args[1]);
44             }
45             catch(UnknownHostException e) {
46                 error("Unknown host exception");
47             }
48
49             info("Starting RTP reception ...");
50
51             if(!receiver.initialize()) {
52                 error("Failed to start RTP receiver");
53             }
54
55             // Wait for a user interruption
56             try {
57                 BufferedReader d =
58                     new BufferedReader(new InputStreamReader(System.in));
59                 String line;
60
61                 while(((line = d.readLine()) != null) && (!receiver.isDone())) {
62                 }
63                 if(line == null) {
64                     info("Ctrl-d key pressed");
65                     info("Stopping RTP receiver");
66                     receiver.close();
67                 }
68                 d.close();
69             }
70             catch(IOException e) {
71                 error("I/O exception on standard input");
72             }
73         }
74     }
75
76     // Print program usage on standard output
77     static void prUsage() {
```

```
78          info("Usage:");
79          info("java Client.ClientMain <RTPRecPort> <pluginType>");
80          info("     <RTPRecPort>: RTP reception port");
81          info("     <pluginType>: plugin short name");
82          System.exit(0);
83      }
84
85      static private void info(String msg) {
86          System.out.println("[INFO] Server: " + msg);
87      }
88
89      static private void error(String msg) {
90          System.err.println("[ERROR] Server: " + msg);
91          System.exit(1);
92      }
93  }
```

Simulator/Server/ServerRTPReceiver.java

```
1  package Server;
2
3  import java.io.*;
4  import java.awt.*;
5  import java.net.*;
6  import java.awt.event.*;
7  import java.util.Vector;
8
9  import javax.media.*;
10 import javax.media.rtp.*;
11 import javax.media.rtp.event.*;
12 import javax.media.rtp.rtcp.*;
13 import javax.media.protocol.*;
14 import javax.media.protocol.DataSource;
15 import javax.media.format.AudioFormat;
16 import javax.media.format.VideoFormat;
17 import javax.media.Format;
18 import javax.media.format.FormatChangeEvent;
19 import javax.media.control.BufferControl;
20 import javax.media.control.TrackControl;
21 import javax.media.control.QualityControl;
22
23 import Server.Plugin.*;
24
25 public class ServerRTPReceiver implements ReceiveStreamListener, SessionListener,
26                                           ControllerListener
27 {
28     // IP address and UDP port of the host from which the RTP data arrives
29     private String ipSrc;
30     private int portSrc;
31
32     // The specified plugin
33     private String pluginType;
34
35     // RTP manager used to receive RTP data
36     RTPManager mgr = null;
37
38     // Vector containing the GUI of all the players
39     Vector playerWindows = null;
40
41     // Convenience variable and object used to manage the RTP data waiting loop
42     boolean dataReceived = false;
43     Object dataSync = new Object();
44
45     // Constructor
46     public ServerRTPReceiver(String ipSrc, int portSrc, String pluginType) {
47         this.ipSrc = ipSrc;
48         this.portSrc = portSrc;
49         this.pluginType = pluginType;
50     }
51
52     // Initialize a RTPManager and open the RTP session
53     protected boolean initialize() {
54         try {
55             playerWindows = new Vector();
56
57             // Open the RTP session
58             info("Open RTP session for: addr: " + ipSrc + " port: " + portSrc);
59
60             mgr = (RTPManager) RTPManager.newInstance();
61             mgr.addSessionListener(this);
62             mgr.addReceiveStreamListener(this);
63
64             // Initialize the RTPManager with the RTPSocketAdapter
65             mgr.initialize(new ServerRTPSocketAdapter(InetAddress.getByName(ipSrc),
66                                                       portSrc,1));
67
68             // Set the input buffer size
69             BufferControl bc =
70                 (BufferControl)mgr.getControl("javax.media.control.BufferControl");
71             if (bc != null)
72                 bc.setBufferLength(1000);
73
74         } catch (Exception e){
75             error("Cannot create the RTP session: " + e.getMessage());
76             close();
77             return false;
78         }
79
80         // Wait for data to arrive
```

```
81          long then = System.currentTimeMillis();
82          // Wait for a maximum of 300 secs
83          long waitingPeriod = 300000;
84
85          try{
86              synchronized (dataSync) {
87                  while (!dataReceived &&
88                          System.currentTimeMillis() - then < waitingPeriod) {
89                      if (!dataReceived)
90                          info("  - Waiting for RTP data to arrive...");
91                      dataSync.wait(1000);
92                  }
93              }
94          } catch (Exception e) {}
95
96          if (!dataReceived) {
97              error("No RTP data were received.");
98              close();
99              return false;
100         }
101         return true;
102     }
103
104     // Return true if all the players are closed
105     public boolean isDone() {
106         return playerWindows.size() == 0;
107     }
108
109     // Close the players and the session manager
110     protected void close() {
111         for(int i = 0; i < playerWindows.size(); i++) {
112             try {
113                 ((PlayerWindow)playerWindows.elementAt(i)).close();
114             } catch (Exception e) {}
115         }
116
117         playerWindows.removeAllElements();
118
119         // Close the RTP session
120         if (mgr != null) {
121             mgr.removeTargets("Closing session");
122             mgr.dispose();
123             mgr = null;
124         }
125     }
126
127     // Return the player window associated to a given player
128     PlayerWindow find(Player p) {
129         for (int i = 0; i < playerWindows.size(); i++) {
130             PlayerWindow pw = (PlayerWindow)playerWindows.elementAt(i);
131             if (pw.player == p)
132                 return pw;
133         }
134         return null;
135     }
136
137     // Return the player window associated to a given RTP receive stream
138     PlayerWindow find(ReceiveStream strm) {
139         for (int i = 0; i < playerWindows.size(); i++) {
140             PlayerWindow pw = (PlayerWindow)playerWindows.elementAt(i);
141             if (pw.stream == strm)
142                 return pw;
143         }
144         return null;
145     }
146
147     // Session listener
148     public synchronized void update(SessionEvent evt) {
149         if (evt instanceof NewParticipantEvent) {
150             Participant p = ((NewParticipantEvent)evt).getParticipant();
151             info("A new participant had just joined: " + p.getCNAME());
152         }
153     }
154
155     // Receive stream listener
156     public synchronized void update(ReceiveStreamEvent evt) {
157
158         RTPManager mgr = (RTPManager)evt.getSource();
159         Participant participant = evt.getParticipant();
160         ReceiveStream stream = evt.getReceiveStream();
161
162         if (evt instanceof RemotePayloadChangeEvent) {
163
```

```
164             info("Received a RTP PayloadChangeEvent");
165             error("Cannot handle payload change");
166             System.exit(1);
167         }
168 
169         else if (evt instanceof NewReceiveStreamEvent) {
170 
171             try {
172                 stream = ((NewReceiveStreamEvent)evt).getReceiveStream();
173                 DataSource ds = stream.getDataSource();
174 
175                 // Find out the formats
176                 RTPControl ctl =
177                     (RTPControl)ds.getControl("javax.media.rtp.RTPControl");
178                 if (ctl != null){
179                     info("Received new RTP stream: " + ctl.getFormat());
180                 } else
181                     info("Received new RTP stream");
182 
183                 if (participant == null)
184                     info("The sender of this stream had yet to be identified");
185                 else {
186                     info("The stream comes from: " + participant.getCNAME());
187                 }
188 
189                 // Try to create a processor to handle the input media locator
190                 Processor processor = null;
191                 try {
192                     processor = javax.media.Manager.createProcessor(ds);
193                 } catch (NoProcessorException npe) {
194                     error("Couldn't create processor");
195                     System.exit(1);
196                 } catch (IOException ioe) {
197                     error("I/O exception while creating processor");
198                     System.exit(1);
199                 }
200 
201                 // Wait for it to be in configured state
202                 boolean result = waitForState(processor, Processor.Configured);
203                 if (result == false) {
204                     error("Couldn't configure processor");
205                     System.exit(1);
206                 }
207 
208                 // Get the tracks from the processor
209                 TrackControl [] tracks = processor.getTrackControls();
210                 if (tracks == null || tracks.length < 1) {
211                     error("Couldn't find tracks in processor");
212                     System.exit(1);
213                 }
214 
215                 Format supported[];
216                 Format chosen;
217                 boolean atLeastOneTrack = false;
218 
219                 // Program the tracks.
220                 for (int i = 0; i < tracks.length; i++) {
221                     Format format = tracks[i].getFormat();
222 
223                     Codec plugin = null;
224 
225                     // Select the specified plugin
226                     if(pluginType.equals("Basic"))
227                         plugin = new H263VideoVerifierBasic();
228                     else if(pluginType.equals("Simple"))
229                         plugin = new H263VideoVerifierSimple();
230                     else if(pluginType.equals("EMSS"))
231                         plugin = new H263VideoVerifierEMSS();
232                     else if(pluginType.equals("Star"))
233                         plugin = new H263VideoVerifierStar();
234                     else if(pluginType.equals("Tree"))
235                         plugin = new H263VideoVerifierTree();
236                     else
237                         error("Bad plugin type");
238 
239                     if(format instanceof VideoFormat) {
240                         try {
241                             info("Setting decoding plugins chain for track "
242                                  + i + " ...");
243                             tracks[i].setCodecChain(new Codec[] {plugin});
244                         }
245                         catch (NotConfiguredError e) {
246                             error("Not Configured Error");
```

```
247                         }
248                         catch (UnsupportedPlugInException e) {
249                             error("Unsupported PlugIn Exception");
250                         }
251                     }
252
253                     if (tracks[i].isEnabled()) {
254
255                         supported = tracks[i].getSupportedFormats();
256
257                         if (supported.length > 0) {
258                             if (supported[0] instanceof VideoFormat) {
259                                 chosen = checkForVideoSizes(tracks[i].getFormat(),
260                                                             supported[0]);
261                             } else
262                                 chosen = supported[0];
263                             tracks[i].setFormat(chosen);
264                             info("Track " + i + " is set to transmit as:");
265                             info("  " + chosen);
266                             atLeastOneTrack = true;
267                         } else
268                             tracks[i].setEnabled(false);
269                       } else
270                       tracks[i].setEnabled(false);
271                 }
272
273                 if (!atLeastOneTrack) {
274                     error("Couldn't set any of the tracks to a valid RTP format");
275                     System.exit(1);
276                 }
277
278                 // Realize the processor
279                 result = waitForState(processor, Controller.Realized);
280                 if (result == false) {
281                     error("Couldn't realize processor");
282                     System.exit(1);
283                 }
284
285                 // Get the output data source of the processor
286                 DataSource pds = processor.getDataOutput();
287
288                 // Create a player by passing datasource to the Media Manager
289                 Player p = javax.media.Manager.createPlayer(pds);
290                 if (p == null) {
291                     error("Couldn't create player");
292                     System.exit(1);
293                 }
294
295                 p.addControllerListener(this);
296                 p.realize();
297                 PlayerWindow pw = new PlayerWindow(p, stream);
298                 playerWindows.addElement(pw);
299
300                 // Start the processor
301                 processor.start();
302
303                 // Notify intialize() that a new stream had arrived.
304                 synchronized (dataSync) {
305                     dataReceived = true;
306                     dataSync.notifyAll();
307                 }
308
309             } catch (Exception e) {
310                 error("New receive stream event exception " + e.getMessage());
311                 System.exit(1);
312             }
313         }
314
315         else if (evt instanceof StreamMappedEvent) {
316             if (stream != null && stream.getDataSource() != null) {
317                 DataSource ds = stream.getDataSource();
318                 // Find out the formats
319                 RTPControl ctl =
320                     (RTPControl)ds.getControl("javax.media.rtp.RTPControl");
321                 info("The previously unidentified stream ");
322                 if (ctl != null)
323                     info("      " + ctl.getFormat());
324                 info("had now been identified as sent by: "
325                      + participant.getCNAME());
326             }
327         }
328
329         else if (evt instanceof TimeoutEvent) {
```

```
330             info("" + participant.getCNAME() + " leaved (time out)");
331             PlayerWindow pw = find(stream);
332             if (pw != null) {
333                 pw.close();
334                 playerWindows.removeElement(pw);
335             }
336             System.exit(0);
337         }
338 
339         else if (evt instanceof ByeEvent) {
340 
341              info("Got \"bye\" from: " + participant.getCNAME());
342              PlayerWindow pw = find(stream);
343              if (pw != null) {
344                 pw.close();
345                 playerWindows.removeElement(pw);
346              }
347         }
348     }
349 
350 
351     // Controller listener for the Players
352     public synchronized void controllerUpdate(ControllerEvent ce) {
353 
354         Player p = (Player)ce.getSourceController();
355 
356         if (p == null)
357             return;
358 
359         if (ce instanceof RealizeCompleteEvent) {
360             PlayerWindow pw = find(p);
361             if (pw == null) {
362                 error("Internal error!");
363                 System.exit(1);
364             }
365             pw.initialize();
366             pw.setVisible(true);
367             p.start();
368         }
369 
370         if (ce instanceof ControllerErrorEvent) {
371             p.removeControllerListener(this);
372             PlayerWindow pw = find(p);
373             if (pw != null) {
374                 pw.close();
375                 playerWindows.removeElement(pw);
376             }
377             error("Internal error: " + ce);
378         }
379     }
380 
381     private void info(String msg) {
382         System.out.println("[INFO] Client RTP Receiver: " + msg);
383     }
384 
385     private void error(String msg) {
386         System.err.println("[ERROR] Client RTP Receiver: " + msg);
387     }
388 
389     // H263 only works for particular sizes
390     Format checkForVideoSizes(Format original, Format supported) {
391 
392         int width, height;
393         Dimension size = ((VideoFormat)original).getSize();
394         Format jpegFmt = new Format(VideoFormat.JPEG_RTP);
395         Format h263Fmt = new Format(VideoFormat.H263_RTP);
396 
397         if(supported.matches(h263Fmt)) {
398             // H263 only supports specific dimensions
399             if (size.width < 128) {
400                 width = 128;
401                 height = 96;
402             } else if (size.width < 176) {
403                 width = 176;
404                 height = 144;
405             } else {
406                 width = 352;
407                 height = 288;
408             }
409         } else
410             return supported;
411 
412         return (new VideoFormat(null,
```

```
413                                 new Dimension(width, height),
414                                 Format.NOT_SPECIFIED,
415                                 null,
416                                 Format.NOT_SPECIFIED)).intersects(supported);
417     }
418
419     // Convenience methods used to handle processor's state changes
420     private Integer stateLock = new Integer(0);
421     private boolean failed = false;
422
423     Integer getStateLock() {
424         return stateLock;
425     }
426
427     void setFailed() {
428         failed = true;
429     }
430
431     private synchronized boolean waitForState(Processor p, int state) {
432         p.addControllerListener(new StateListener());
433         failed = false;
434
435         // Call the required method on the processor
436         if (state == Processor.Configured) {
437             p.configure();
438         } else if (state == Processor.Realized) {
439             p.realize();
440         }
441
442         while (p.getState() < state && !failed) {
443             synchronized (getStateLock()) {
444                 try {
445                     getStateLock().wait();
446                 } catch (InterruptedException ie) {
447                     return false;
448                 }
449             }
450         }
451
452         if (failed)
453             return false;
454         else
455             return true;
456     }
457
458     // Internal classes
459
460     // Convenience class used to handle processor's state changes
461     class StateListener implements ControllerListener {
462
463         public void controllerUpdate(ControllerEvent ce) {
464
465             // If there was an error during configure or
466             // realize, the processor will be closed
467             if (ce instanceof ControllerClosedEvent)
468                 setFailed();
469
470             // All controller events, send a notification
471             // to the waiting thread in waitForState method
472             if (ce instanceof ControllerEvent) {
473                 synchronized (getStateLock()) {
474                     getStateLock().notifyAll();
475                 }
476             }
477         }
478     }
479
480     // GUI classes for the Player
481     class PlayerWindow extends Frame {
482
483         Player player;
484         ReceiveStream stream;
485
486         PlayerWindow(Player p, ReceiveStream strm) {
487             player = p;
488             stream = strm;
489         }
490
491         public void initialize() {
492             add(new PlayerPanel(player));
493         }
494
495         public void close() {
```

```
496             player.close();
497             setVisible(false);
498             dispose();
499         }
500
501         public void addNotify() {
502             super.addNotify();
503             pack();
504         }
505     }
506
507     class PlayerPanel extends Panel {
508
509         Component vc, cc;
510
511         PlayerPanel(Player p) {
512             setLayout(new BorderLayout());
513             if ((vc = p.getVisualComponent()) != null)
514                 add("Center", vc);
515             if ((cc = p.getControlPanelComponent()) != null)
516                 add("South", cc);
517         }
518
519         public Dimension getPreferredSize() {
520             int w = 0, h = 0;
521             if (vc != null) {
522                 Dimension size = vc.getPreferredSize();
523                 w = size.width;
524                 h = size.height;
525             }
526             if (cc != null) {
527                 Dimension size = cc.getPreferredSize();
528                 if (w == 0)
529                     w = size.width;
530                 h += size.height;
531             }
532             if (w < 160)
533                 w = 160;
534             return new Dimension(w, h);
535         }
536     }
537 }
```

Simulator/Server/ServerRTPSocketAdapter.java

The content of this file is the same as the content of the prototype client file `Client/ClientRTPSocketAdapter.java` printed in section A.1.1.

Simulator/Server/Plugin/H263VideoVerifierBasic.java

```
1  package Server.Plugin;
2
3  import java.io.*;
4  import java.util.*;
5  import javax.media.*;
6  import javax.media.format.*;
7  import javax.media.format.AudioFormat;
8  import java.security.*;
9
10 public class H263VideoVerifierBasic implements Codec {
11
12     private static String CodecName="H263VideoVerifierBasic";
13
14     /** chosen input Format **/
15     protected VideoFormat inputFormat;
16
17     /** chosen output Format **/
18     protected VideoFormat outputFormat;
19
20     /** supported input Formats **/
21     protected Format[] supportedInputFormats=new Format[0];
22
23     /** supported output Formats **/
24     protected Format[] supportedOutputFormats=new Format[0];
25
26     // Buffer number
27     int buffNbr;
28
29     // JCA variables
30     PublicKey pubKey;
31     Signature dsa;
32
33     /** initialize the plugin **/
34     public H263VideoVerifierBasic() {
35
36         // Set the input and output formats of this plugin
37         supportedInputFormats = new Format[] {
38             new VideoFormat(VideoFormat.H263_RTP)
39         };
40         supportedOutputFormats = new Format[] {
41             new VideoFormat(VideoFormat.H263_RTP)
42         };
43
44         // Read the public key
45         File f = new File("./pubKey.data");
46
47         FileInputStream fis = null;
48
49         try {
50             fis = new FileInputStream(f);
51         }
52         catch(FileNotFoundException e) {
53             error("File Not Found Exception");
54         }
55
56         try {
57             ObjectInput ois = new ObjectInputStream(fis);
58
59             try {
60                 pubKey = (PublicKey) ois.readObject();
61             }
62             catch(ClassNotFoundException e) {
63                 error("Class Not Found Exception");
64             }
65
66             ois.close();
67
68             info("Public Key File read");
69         }
70         catch(IOException e) {
71             error("I/O Exception");
```

```
72          }
73
74          // Signature verification initialization
75          try {
76              dsa = Signature.getInstance("SHA1withDSA");
77              dsa.initVerify(pubKey);
78          }
79          catch(NoSuchAlgorithmException e) {
80              error("No Such Algorithm Exception");
81          }
82          catch(InvalidKeyException e) {
83              error("Invalid Key Exception");
84          }
85      }
86
87      /** get the resources needed by this plugin **/
88      public void open() throws ResourceUnavailableException {
89      }
90
91      /** free the resources allocated by this plugin **/
92      public void close() {
93      }
94
95      /** reset the plugin **/
96      public void reset() {
97      }
98
99      /** no controls for this simple plugin **/
100     public Object[] getControls() {
101         return (Object[]) new Control[0];
102     }
103
104     /** Return the control based on a control type for the plugin**/
105     public Object getControl(String controlType) {
106         try {
107             Class cls = Class.forName(controlType);
108             Object cs[] = getControls();
109             for (int i = 0; i < cs.length; i++) {
110                 if (cls.isInstance(cs[i]))
111                     return cs[i];
112             }
113             return null;
114         } catch (Exception e) { // no such controlType or such control
115             return null;
116         }
117     }
118
119     /************** format methods *************/
120     /** set the input format **/
121     public Format setInputFormat(Format input) {
122         // the following code assumes valid Format
123         inputFormat = (VideoFormat)input;
124         return (Format)inputFormat;
125     }
126
127     /** set the output format **/
128     public Format setOutputFormat(Format output) {
129         // the following code assumes valid Format
130         outputFormat = (VideoFormat)output;
131         return (Format)outputFormat;
132     }
133
134     /** get the input format **/
135     protected Format getInputFormat() {
136         return inputFormat;
137     }
138
139     /** get the output format **/
140     protected Format getOutputFormat() {
141         return outputFormat;
142     }
143
144     /** supported input formats **/
145     public Format [] getSupportedInputFormats() {
146         return supportedInputFormats;
147     }
148
149     /** output Formats for the selected input format **/
150     public Format [] getSupportedOutputFormats(Format in) {
151         if (in == null)
152             return supportedOutputFormats;
153         else {
154             Format outs[] = new Format[1];
```

```
155             outs[0] = in;
156             return outs;
157         }
158     }
159 
160     /** return plugin name **/
161     public String getName() {
162         return CodecName;
163     }
164 
165     /** do the processing **/
166     public int process(Buffer inputBuffer, Buffer outputBuffer){
167 
168         // Get the input buffer info
169         int inLength = inputBuffer.getLength();
170         int inOffset = inputBuffer.getOffset();
171         byte[] inData =
172             getValidData((byte[]) inputBuffer.getData(),inOffset,inLength);
173 
174         // Retrieve the signature from the buffer
175         int sigSize = getSigSize(inData);
176         buffNbr = extractBuffNbr(inData,sigSize);
177 
178         // Print some info
179         info("*****************************");
180         info("Buffer #" + buffNbr);
181         info("Input buffer length: " + inLength);
182         info("Format/encoding : " + inputBuffer.getFormat().getEncoding());
183 
184         info("Signature present, length = " + sigSize);
185 
186         // Signature verification
187         if(verifSignature(inData,extractSignature(inData,sigSize)))
188             info("Signature verification ok");
189         else
190             error("Bad signature");
191 
192         // Compute the length of the video data contained in this buffer
193         int videoDataLength = inLength - 1 - sigSize - 4;
194 
195         // Extract the video data from this buffer
196         byte[] outData = extractVideoData(inData,videoDataLength);
197 
198         // Set the content and properties of the output buffer
199         outputBuffer.setData(outData);
200         outputBuffer.setFormat(inputBuffer.getFormat());
201         outputBuffer.setLength(outData.length);
202         outputBuffer.setOffset(0);
203 
204         return BUFFER_PROCESSED_OK;
205     }
206 
207     // Extract the buffer number
208     private int extractBuffNbr(byte[] data, int sigSize) {
209 
210         byte[] tmp = new byte[4];
211         int i;
212         int offset = data.length - 1 - sigSize - 4;
213 
214         for(i=0;i<4;i++)
215             tmp[i] = data[offset + i];
216 
217         return byteArray2Int(tmp);
218     }
219 
220     // Convert a byte array to an integer
221     private int byteArray2Int(byte[] val) {
222         int tmp = 0;
223         int i;
224         for (i=3;i>0;i--) {
225             tmp = tmp + ((int) val[i] & (int) 0xff);
226             tmp = tmp << 8;
227         }
228         tmp = tmp + ((int) val[0] & (int) 0xff);
229         return tmp;
230     }
231 
232     // Extract the video data from a buffer
233     private byte[] extractVideoData(byte[] data, int length) {
234 
235         byte[] tmp = new byte[length];
236         int i;
237 
```

```
238         for(i=0;i<length;i++)
239             tmp[i] = data[i];
240 
241         return tmp;
242     }
243 
244     // Compute the signature size
245     private int getSigSize(byte[] data) {
246         return (int) (data[data.length - 1] & 0x7f);
247     }
248 
249     // Extract the signature from a buffer
250     private byte[] extractSignature(byte[] data,int sigSize) {
251 
252         byte[] sig = new byte[sigSize];
253         int offset = data.length - sigSize - 1;
254         int i;
255         for(i=0;i<sigSize;i++)
256             sig[i] = data[offset + i];
257         return sig;
258     }
259 
260     // Verify the signature of a buffer
261     private boolean verifSignature(byte[] data, byte[] sig) {
262 
263         byte[] tmp = new byte[data.length - sig.length - 1];
264 
265         int i;
266         boolean result = false;
267 
268         for(i=0;i<tmp.length;i++)
269             tmp[i] = data[i];
270 
271         try {
272             dsa.update(tmp);
273             result = dsa.verify(sig);
274         }
275         catch(SignatureException e) {
276             error("Signature Exception");
277         }
278 
279         return result;
280     }
281 
282     // Extract the valid video data from the input buffer
283     private byte[] getValidData(byte[] data,int offset,int length) {
284 
285         byte[] tmp = new byte[length];
286         int i;
287 
288         for(i=0;i<length;i++)
289             tmp[i] = data[offset + i];
290 
291         return tmp;
292     }
293 
294     private void info(String msg) {
295         System.out.println("[INFO] H263 Video Verifier Plugin: " + msg);
296     }
297 
298     private void error(String msg) {
299         System.err.println("[ERROR] H263 Video Verifier Plugin: " + msg);
300         System.exit(1);
301     }
302 }
```

Simulator/Server/Plugin/H263VideoVerifierSimple.java

```
1  package Server.Plugin;
2
3  import java.io.*;
4  import java.util.*;
5  import javax.media.*;
6  import javax.media.format.*;
7  import javax.media.format.AudioFormat;
8  import java.security.*;
9
10 public class H263VideoVerifierSimple implements Codec {
11
12     private static String CodecName="H263VideoVerifierSimple";
13
14     /** chosen input Format **/
15     protected VideoFormat inputFormat;
16
17     /** chosen output Format **/
18     protected VideoFormat outputFormat;
19
20     /** supported input Formats **/
21     protected Format[] supportedInputFormats=new Format[0];
22
23     /** supported output Formats **/
24     protected Format[] supportedOutputFormats=new Format[0];
25
26     // Buffer number
27     int buffNbr;
28
29     // Hash length
30     int hashSize = 16;
31
32     // A temporary hash
33     byte[] hashTmp = null;
34
35     // True if the current buffer is the first of a chain
36     // In this case, no hash can be tested
37     boolean chainBeginning = true;
38
39     // The hasher
40     MessageDigest digester = null;
41
42     // JCA variables
43     PublicKey pubKey;
44     Signature dsa;
45
46     /** initialize the plugin **/
47     public H263VideoVerifierSimple() {
48
49         // Set the input and output formats of this plugin
50         supportedInputFormats = new Format[] {
51             new VideoFormat(VideoFormat.H263_RTP)
52         };
53         supportedOutputFormats = new Format[] {
54             new VideoFormat(VideoFormat.H263_RTP)
55         };
56
57         // Initialize the hasher
58         try {
59             digester = MessageDigest.getInstance("MD5");
60         }
61         catch(NoSuchAlgorithmException e) {
62             error("No Such Algorithm Exception");
63         }
64
65         // Read the public key
66         File f = new File("./pubKey.data");
67
68         FileInputStream fis = null;
69
70         try {
71             fis = new FileInputStream(f);
72         }
73         catch(FileNotFoundException e) {
74             error("File Not Found Exception");
75         }
76
77         try {
78             ObjectInput ois = new ObjectInputStream(fis);
79
80             try {
```

```
81                  pubKey = (PublicKey) ois.readObject();
82              }
83              catch(ClassNotFoundException e) {
84                  error("Class Not Found Exception");
85              }
86
87              ois.close();
88
89              info("Public Key File read");
90          }
91          catch(IOException e) {
92              error("I/O Exception");
93          }
94
95          // Signature verification initialization
96          try {
97              dsa = Signature.getInstance("SHA1withDSA");
98              dsa.initVerify(pubKey);
99          }
100         catch(NoSuchAlgorithmException e) {
101             error("No Such Algorithm Exception");
102         }
103         catch(InvalidKeyException e) {
104             error("Invalid Key Exception");
105         }
106     }
107
108     /** get the resources needed by this plugin **/
109     public void open() throws ResourceUnavailableException {
110     }
111
112     /** free the resources allocated by this plugin **/
113     public void close() {
114     }
115
116     /** reset the plugin **/
117     public void reset() {
118     }
119
120     /** no controls for this simple plugin **/
121     public Object[] getControls() {
122         return (Object[]) new Control[0];
123     }
124
125     /** Return the control based on a control type for the plugin **/
126     public Object getControl(String controlType) {
127         try {
128             Class cls = Class.forName(controlType);
129             Object cs[] = getControls();
130             for (int i = 0; i < cs.length; i++) {
131                 if (cls.isInstance(cs[i]))
132                     return cs[i];
133             }
134             return null;
135         } catch (Exception e) { // no such controlType or such control
136             return null;
137         }
138     }
139
140     /************** format methods *************/
141     /** set the input format **/
142     public Format setInputFormat(Format input) {
143         // the following code assumes valid Format
144         inputFormat = (VideoFormat)input;
145         return (Format)inputFormat;
146     }
147
148     /** set the output format **/
149     public Format setOutputFormat(Format output) {
150         // the following code assumes valid Format
151         outputFormat = (VideoFormat)output;
152         return (Format)outputFormat;
153     }
154
155     /** get the input format **/
156     protected Format getInputFormat() {
157         return inputFormat;
158     }
159
160     /** get the output format **/
161     protected Format getOutputFormat() {
162         return outputFormat;
163     }
```

```
164
165     /** supported input formats **/
166     public Format [] getSupportedInputFormats() {
167         return supportedInputFormats;
168     }
169
170     /** output Formats for the selected input format **/
171     public Format [] getSupportedOutputFormats(Format in) {
172         if (in == null)
173             return supportedOutputFormats;
174         else {
175             Format outs[] = new Format[1];
176             outs[0] = in;
177             return outs;
178         }
179     }
180
181     /** return plugin name **/
182     public String getName() {
183         return CodecName;
184     }
185
186     /** do the processing **/
187     public int process(Buffer inputBuffer, Buffer outputBuffer){
188
189         // Get the input buffer info
190         int inLength = inputBuffer.getLength();
191         int inOffset = inputBuffer.getOffset();
192         byte[] inData =
193             getValidData((byte[]) inputBuffer.getData(),inOffset,inLength);
194
195         // Retrieve the signature size from the buffer
196         int sigSize = getSigSize(inData);
197
198         // Retrieve the buffer number
199         buffNbr = extractBuffNbr(inData,sigSize);
200
201         // Print some info
202         info("*******************************");
203         info("Buffer #" + buffNbr);
204         info("Input buffer length: " + inLength);
205         info("Format/encoding : " + inputBuffer.getFormat().getEncoding());
206
207         // Hash verification
208         if(chainBeginning) {
209             info("First chain hash can't be tested");
210             chainBeginning = false;
211         }
212         else
213             if(verifHash(extractHash(inData,sigSize)))
214                 info("Hash verification ok");
215             else
216                 info("ERROR: Bad hash");
217
218         if(sigSize == 0) {
219             info("No signature");
220             // Compute hash
221             hashTmp = computeHash(inData);
222         }
223         else {
224             info("Signature present, length = " + sigSize);
225             // Signature verification
226             if(verifSignature(inData,extractSignature(inData,sigSize)))
227                 info("Signature verification ok");
228             else
229                 error("Bad signature");
230             chainBeginning = true;
231         }
232
233         // Compute the length of the video data contained in this buffer
234         int videoDataLength = inLength - 1 - sigSize - hashSize - 4;
235
236         // Extract the video data from this buffer
237         byte[] outData = extractVideoData(inData,videoDataLength);
238
239         // Set the content and properties of the output buffer
240         outputBuffer.setData(outData);
241         outputBuffer.setFormat(inputBuffer.getFormat());
242         outputBuffer.setLength(outData.length);
243         outputBuffer.setOffset(0);
244
245         return BUFFER_PROCESSED_OK;
246     }
```

```
247
248     // Extract the buffer number
249     private int extractBuffNbr(byte[] data, int sigSize) {
250
251         byte[] tmp = new byte[4];
252         int i;
253         int offset = data.length - 1 - sigSize - hashSize - 4;
254
255         for(i=0;i<4;i++)
256             tmp[i] = data[offset + i];
257
258         return byteArray2Int(tmp);
259     }
260
261     // Convert a byte array to an integer
262     private int byteArray2Int(byte[] val) {
263         int tmp = 0;
264         int i;
265         for (i=3;i>0;i--) {
266             tmp = tmp + ((int) val[i] & (int) 0xff);
267             tmp = tmp << 8;
268         }
269         tmp = tmp + ((int) val[0] & (int) 0xff);
270         return tmp;
271     }
272
273     // Extract the video data from a buffer
274     private byte[] extractVideoData(byte[] data, int length) {
275
276         byte[] tmp = new byte[length];
277         int i;
278
279         for(i=0;i<length;i++)
280             tmp[i] = data[i];
281
282         return tmp;
283     }
284
285     // Compute the signature size
286     private int getSigSize(byte[] data) {
287         if((data[data.length - 1] & 0x80) == 0)
288             return 0;
289         else
290             return (int) (data[data.length - 1] & 0x7f);
291     }
292
293     // Extract the signature from a buffer
294     private byte[] extractSignature(byte[] data,int sigSize) {
295
296         byte[] sig = new byte[sigSize];
297         int offset = data.length - sigSize - 1;
298         int i;
299         for(i=0;i<sigSize;i++)
300             sig[i] = data[offset + i];
301         return sig;
302     }
303
304     // Verify the signature of a buffer
305     private boolean verifSignature(byte[] data, byte[] sig) {
306
307         byte[] tmp = new byte[data.length - sig.length - 1];
308
309         int i;
310         boolean result = false;
311
312         for(i=0;i<tmp.length;i++)
313             tmp[i] = data[i];
314
315         try {
316             dsa.update(tmp);
317             result = dsa.verify(sig);
318         }
319         catch(SignatureException e) {
320             error("Signature Exception");
321         }
322
323         return result;
324     }
325
326     // Extract the hash from a buffer
327     private byte[] extractHash(byte[] data, int sigSize) {
328
329         byte[] tmp = new byte[hashSize];
```

```
330         int i;
331         int offset = data.length - 1 - sigSize - hashSize;
332 
333         for(i=0;i<hashSize;i++)
334             tmp[i] = data[offset + i];
335 
336         return tmp;
337     }
338 
339     // Return true if and only if hash equals hashTmp
340     private boolean verifHash(byte[] hash) {
341         int i;
342         for(i=0;i<hashSize;i++)
343             if(hash[i] != hashTmp[i])
344                 return false;
345         return true;
346     }
347 
348     // Compute current buffer hash
349     private byte[] computeHash(byte[] data) {
350         return digester.digest(data);
351     }
352 
353     // Extract the valid video data from the input buffer
354     private byte[] getValidData(byte[] data,int offset,int length) {
355 
356         byte[] tmp = new byte[length];
357         int i;
358 
359         for(i=0;i<length;i++)
360             tmp[i] = data[offset + i];
361 
362         return tmp;
363     }
364 
365     private void info(String msg) {
366         System.out.println("[INFO] H263 Video Verifier Plugin: " + msg);
367     }
368 
369     private void error(String msg) {
370         System.err.println("[ERROR] H263 Video Verifier Plugin: " + msg);
371         System.exit(1);
372     }
373 }
```

Simulator/Server/Plugin/H263VideoVerifierEMSS.java

```
1  package Server.Plugin;
2
3  import java.io.*;
4  import java.util.*;
5  import javax.media.*;
6  import javax.media.format.*;
7  import javax.media.format.AudioFormat;
8  import java.security.*;
9
10 public class H263VideoVerifierEMSS implements Codec {
11
12     private static String CodecName="H263VideoVerifierEMSS";
13
14     /** chosen input Format **/
15     protected VideoFormat inputFormat;
16
17     /** chosen output Format **/
18     protected VideoFormat outputFormat;
19
20     /** supported input Formats **/
21     protected Format[] supportedInputFormats=new Format[0];
22
23     /** supported output Formats **/
24     protected Format[] supportedOutputFormats=new Format[0];
25
26     // Buffer number
27     int buffNbr;
28
29     // Hash distribution scheme parameters
30     int schemeLength = 4; // Such that hashSize % schemeLength = 0
31     int schemeRange = 16;
32
33     // Some constants
34     int hashSize = 16;
35     // Minimum number of valid hash parts to reach
36     // in order to validate a received buffer
37     // quorum <= (hashSize / schemeLength)
38     int quorum = (hashSize / schemeLength);
39
40     // Verification queues
41     // Unchecked hash values queue
42     Vector hashQueue = new Vector();
43     // Hash part queue
44     Vector verifQueue = new Vector();
45
46     // The hasher
47     MessageDigest digester = null;
48
49     // JCA variables
50     PublicKey pubKey;
51     Signature dsa;
52
53     /** initialize the plugin **/
54     public H263VideoVerifierEMSS() {
55
56         // Set the input and output formats of this plugin
57         supportedInputFormats = new Format[] {
58             new VideoFormat(VideoFormat.H263_RTP)
59         };
60         supportedOutputFormats = new Format[] {
61             new VideoFormat(VideoFormat.H263_RTP)
62         };
63
64         // Initialize the hasher
65         try {
66             digester = MessageDigest.getInstance("MD5");
67         }
68         catch(NoSuchAlgorithmException e) {
69             error("No Such Algorithm Exception");
70         }
71
72         // Read the public key
73         File f = new File("./pubKey.data");
74
75         FileInputStream fis = null;
76
77         try {
78             fis = new FileInputStream(f);
79         }
80         catch(FileNotFoundException e) {
```

```
81              error("File Not Found Exception");
82          }
83
84          try {
85              ObjectInput ois = new ObjectInputStream(fis);
86
87              try {
88                  pubKey = (PublicKey) ois.readObject();
89              }
90              catch(ClassNotFoundException e) {
91                  error("Class Not Found Exception");
92              }
93
94              ois.close();
95
96              info("Public Key File read");
97          }
98          catch(IOException e) {
99              error("I/O Exception");
100         }
101
102         // Signature verification initialization
103         try {
104             dsa = Signature.getInstance("SHA1withDSA");
105             dsa.initVerify(pubKey);
106         }
107         catch(NoSuchAlgorithmException e) {
108             error("No Such Algorithm Exception");
109         }
110         catch(InvalidKeyException e) {
111             error("Invalid Key Exception");
112         }
113     }
114
115     /** get the resources needed by this plugin **/
116     public void open() throws ResourceUnavailableException {
117     }
118
119     /** free the resources allocated by this plugin **/
120     public void close() {
121     }
122
123     /** reset the plugin **/
124     public void reset() {
125     }
126
127     /** no controls for this simple plugin **/
128     public Object[] getControls() {
129         return (Object[]) new Control[0];
130     }
131
132     /** Return the control based on a control type for the plugin **/
133     public Object getControl(String controlType) {
134         try {
135             Class cls = Class.forName(controlType);
136             Object cs[] = getControls();
137             for (int i = 0; i < cs.length; i++) {
138                 if (cls.isInstance(cs[i]))
139                     return cs[i];
140             }
141             return null;
142         } catch (Exception e) { // no such controlType or such control
143             return null;
144         }
145     }
146
147     /************** format methods *************/
148     /** set the input format **/
149     public Format setInputFormat(Format input) {
150         // the following code assumes valid Format
151         inputFormat = (VideoFormat)input;
152         return (Format)inputFormat;
153     }
154
155     /** set the output format **/
156     public Format setOutputFormat(Format output) {
157         // the following code assumes valid Format
158         outputFormat = (VideoFormat)output;
159         return (Format)outputFormat;
160     }
161
162     /** get the input format **/
163     protected Format getInputFormat() {
```

```
164         return inputFormat;
165     }
166
167     /** get the output format **/
168     protected Format getOutputFormat() {
169         return outputFormat;
170     }
171
172     /** supported input formats **/
173     public Format [] getSupportedInputFormats() {
174         return supportedInputFormats;
175     }
176
177     /** output Formats for the selected input format **/
178     public Format [] getSupportedOutputFormats(Format in) {
179         if (in == null)
180             return supportedOutputFormats;
181         else {
182             Format outs[] = new Format[1];
183             outs[0] = in;
184             return outs;
185         }
186     }
187
188     /** return plugin name **/
189     public String getName() {
190         return CodecName;
191     }
192
193     /** do the processing **/
194     public int process(Buffer inputBuffer, Buffer outputBuffer){
195
196         // Get the input buffer info
197         int inLength = inputBuffer.getLength();
198         int inOffset = inputBuffer.getOffset();
199         byte[] inData =
200             getValidData((byte[]) inputBuffer.getData(),inOffset,inLength);
201
202         // Retrieve the buffer number
203         buffNbr = extractBuffNbr(inData);
204         // Retrieve the signature size from the buffer
205         int sigSize = getSigSize(inData);
206         // Retrive the number of hash parts present in this buffer
207         int hashesNbr = getHashesNbr(inData,sigSize);
208
209         // Print some info
210         info("*******************************");
211         info("Buffer #" + buffNbr);
212
213         if(sigSize == 0)
214             info("No Signature");
215         else
216             info("Signature present, length: " + sigSize);
217
218         info("Number of hash parts: " + hashesNbr);
219         info("Input buffer length: " + inLength);
220         info("Format/encoding : " + inputBuffer.getFormat().getEncoding());
221
222         byte[] outData;
223         int videoDataLength;
224
225         // Compute the length of the video data contained in this buffer
226         if(sigSize == 0)
227             videoDataLength =
228                 inLength - 5 - (hashesNbr * ((hashSize / schemeLength) + 5));
229         else {
230             videoDataLength =
231                 inLength - 6 - (hashesNbr * ((hashSize / schemeLength) + 5))
232                 - sigSize;
233             if(verifSignature(inData,extractSignature(inData,sigSize)))
234                 info("Signature checking ok");
235             else
236                 error("Bad Signature");
237         }
238         // Extract the video data from this buffer
239         outData = extractVideoData(inData,videoDataLength);
240         // Extract the hash parts from this buffer
241         extractHashPart(inData,videoDataLength,hashesNbr);
242
243         // Add the hash value of the current buffer in hashQueue
244         hashQueue.add(computeHash(inData,sigSize));
245
246         // Check a maximum of hash values from hashQueue
```

```
247         if(verifHash())
248             info("Hash checking ok");
249         else
250             error("Bad Hash");
251 
252         // Remove the unverifiable hash values from hashQueue
253         cleanVerifQueue();
254         info("verifQueue.size = " + verifQueue.size());
255 
256         // Remove the hash parts which are not needed anymore
257         cleanHashQueue();
258         info("hashQueue.size = " + hashQueue.size());
259 
260         // Set the content and properties of the output buffer
261         outputBuffer.setData(outData);
262         outputBuffer.setFormat(inputBuffer.getFormat());
263         outputBuffer.setLength(outData.length);
264         outputBuffer.setOffset(0);
265 
266         return BUFFER_PROCESSED_OK;
267     }
268 
269     // Try to verify a maximum of hash values from hashQueue
270     private boolean verifHash() {
271 
272         HashQueueItem hQItem;
273         VerifQueueItem vQItem;
274         int partSize = hashSize / schemeLength;
275 
276         int i,j,k,offset;
277         for(i=0;i<hashQueue.size();i++) {
278             hQItem = (HashQueueItem) hashQueue.get(i);
279 
280             for(j=0;j<verifQueue.size();j++) {
281                 vQItem = (VerifQueueItem) verifQueue.get(j);
282                 if(hQItem.srcBuffer == vQItem.srcBuffer) {
283                     info("Checking hash #" + hQItem.srcBuffer
284                             + " Part #" + vQItem.partNbr);
285                     offset = vQItem.partNbr * partSize;
286 
287                     for(k=0;k<partSize;k++)
288                         if(vQItem.part[k] != hQItem.hash[offset + k]) {
289                             return false;
290                         }
291                     hQItem.acPartList[vQItem.partNbr] = true;
292                     hashQueue.set(i,hQItem);
293                     verifQueue.removeElementAt(j);
294                     j--;
295                 }
296             }
297         }
298         return true;
299     }
300 
301     // Verify the signature of a buffer
302     private boolean verifSignature(byte[] data, byte[] sig) {
303 
304         byte[] tmp = new byte[data.length - sig.length - 1];
305         boolean result = false;
306 
307         int i,j;
308         int firstMax = data.length - 5 - sig.length;
309 
310         for(i=0;i<firstMax;i++)
311             tmp[i] = data[i];
312 
313         for(j=data.length-4;i<tmp.length;i++,j++)
314             tmp[i] = data[j];
315 
316         try {
317             dsa.update(tmp);
318             result = dsa.verify(sig);
319         }
320         catch(SignatureException e) {
321             error("Signature Exception");
322         }
323 
324         return result;
325     }
326 
327     // Compute the hash value of a buffer
328     private HashQueueItem computeHash(byte[] data, int sigSize) {
329 
```

```
330         HashQueueItem hQItem = new HashQueueItem();
331         hQItem.srcBuffer = buffNbr;
332 
333         if(sigSize == 0) {
334             hQItem.hash = digester.digest(data);
335             return hQItem;
336         }
337 
338         byte[] tmp = new byte[data.length - sigSize - 1];
339 
340         int i,j;
341         int firstMax = data.length - 5 -sigSize;
342         for(i=0;i<firstMax;i++)
343             tmp[i] = data[i];
344 
345         for(j=data.length-4;i<tmp.length;i++,j++)
346             tmp[i] = data[j];
347 
348         hQItem.hash = digester.digest(tmp);
349 
350         return hQItem;
351     }
352 
353     // Remove the unverifiable hashes from hash queue
354     private void cleanHashQueue() {
355 
356         HashQueueItem hQItem;
357         int i,j,acc;
358 
359         for(i=0;i<hashQueue.size();i++) {
360             hQItem = (HashQueueItem) hashQueue.get(i);
361 
362             for(j=0,acc=0;j<hQItem.acPartList.length;j++)
363                 if(hQItem.acPartList[j])
364                     acc++;
365             if(acc >= quorum) {
366                 info("Verified hash #" + hQItem.srcBuffer
367                         + " verified and removed from queue");
368                 hashQueue.removeElementAt(i);
369                 i--;
370             }
371             else if(hQItem.srcBuffer < buffNbr - schemeRange) {
372                 info("Hash #" + hQItem.srcBuffer
373                         + " is too old and has been removed from queue");
374                 hashQueue.removeElementAt(i);
375                 i--;
376             }
377         }
378     }
379 
380     // Remove the unneeded hash parts from the verif queue
381     private void cleanVerifQueue() {
382 
383         VerifQueueItem vQItem;
384         int i;
385 
386         for(i=0;i<verifQueue.size();i++) {
387             vQItem = (VerifQueueItem) verifQueue.get(i);
388 
389             if(vQItem.srcBuffer < buffNbr - schemeRange) {
390                 verifQueue.removeElementAt(i);
391                 i--;
392             }
393         }
394     }
395 
396     // Extract the buffer number
397     private int extractBuffNbr(byte[] data) {
398 
399         byte[] tmp = new byte[4];
400         int i;
401 
402         for(i=0;i<4;i++)
403             tmp[i] = data[data.length - 4 + i];
404 
405         return byteArray2Int(tmp);
406     }
407 
408     // Extract the video data from a buffer
409     private byte[] extractVideoData(byte[] data, int length) {
410 
411         byte[] tmp = new byte[length];
412         int i;
```

```
413
414             for(i=0;i<length;i++)
415                 tmp[i] = data[i];
416
417             return tmp;
418         }
419
420         // Extract the hash parts from a buffer
421         private void extractHashPart(byte[] data, int videoDataLength, int hashNbr) {
422
423             VerifQueueItem tmp;
424             byte[] nbrTmp;
425             int hashPartSize = hashSize / schemeLength;
426             int maxLimit = videoDataLength + (hashNbr * (hashPartSize + 5));
427
428             int i,j,k;
429             for(i=videoDataLength;i<maxLimit;i=i+hashPartSize+5) {
430                 tmp = new VerifQueueItem();
431
432                 tmp.part = new byte[hashPartSize];
433
434                 for(j=0;j<hashPartSize;j++)
435                     tmp.part[j] = data[i+j];
436
437                 tmp.partNbr = (int) data[i+j];
438                 j++;
439
440                 nbrTmp = new byte[4];
441
442                 for(k=0;k<4;k++)
443                     nbrTmp[k] = data[i+j+k];
444
445                 tmp.srcBuffer = byteArray2Int(nbrTmp);
446
447                 verifQueue.add(tmp);
448             }
449         }
450
451         // Extract the signature from a buffer
452         private byte[] extractSignature(byte[] data,int sigSize) {
453
454             byte[] sig = new byte[sigSize];
455             int offset = data.length - sigSize - 5;
456             int i;
457             for(i=0;i<sigSize;i++)
458                 sig[i] = data[offset + i];
459             return sig;
460         }
461
462         // Count the hash values  present in a buffer
463         private int getHashesNbr(byte[] data, int sigSize) {
464             if(sigSize == 0)
465                 return (int) (data[data.length - 5] & 0x7f);
466             else
467                 return (int) (data[data.length - 6 - sigSize] & 0x7f);
468         }
469
470         // Compute the signature size
471         private int getSigSize(byte[] data) {
472             if((data[data.length - 5] & 0x80) == 0)
473                 return 0;
474             else
475                 return (int) (data[data.length - 5] & 0x7f);
476         }
477
478         // Extract the valid video data from the input buffer
479         private byte[] getValidData(byte[] data,int offset,int length) {
480
481             byte[] tmp = new byte[length];
482             int i;
483
484             for(i=0;i<length;i++)
485                 tmp[i] = data[offset + i];
486
487             return tmp;
488         }
489
490         // Convert a byte array to an integer
491         private int byteArray2Int(byte[] val) {
492             int tmp = 0;
493             int i;
494             for (i=3;i>0;i--) {
495                 tmp = tmp + ((int) val[i] & (int) 0xff);
```

```
496             tmp = tmp << 8;
497         }
498         tmp = tmp + ((int) val[0] & (int) 0xff);
499         return tmp;
500     }
501
502     private void info(String msg) {
503         System.out.println("[INFO] H263 Video Verifier Plugin: " + msg);
504     }
505
506     private void error(String msg) {
507         System.err.println("[ERROR] H263 Video Verifier Plugin: " + msg);
508         System.exit(1);
509     }
510
511     // Internal classes
512
513     // Class defining the items contained in hashQueue
514     class HashQueueItem {
515
516         // Source buffer number
517         int srcBuffer;
518
519         // Hash
520         byte[] hash;
521
522         // List of the already checked parts
523         boolean[] acPartList = new boolean[hashSize / schemeLength];
524
525         public HashQueueItem() {
526             int i;
527             for(i=0;i<acPartList.length;i++)
528                 acPartList[i] = false;
529         }
530     }
531
532     // Class defining the items contained in verifQueue
533     class VerifQueueItem {
534
535         // Source buffer number
536         int srcBuffer;
537
538         // Part number
539         int partNbr;
540
541         // Part
542         byte[] part;
543     }
544 }
```

Simulator/Server/Plugin/H263VideoVerifierStar.java

```
1  package Server.Plugin;
2
3  import java.io.*;
4  import java.util.*;
5  import javax.media.*;
6  import javax.media.format.*;
7  import javax.media.format.AudioFormat;
8  import java.security.*;
9
10 public class H263VideoVerifierStar implements Codec {
11
12     private static String CodecName="H263VideoVerifierStar";
13
14     /** chosen input Format **/
15     protected VideoFormat inputFormat;
16
17     /** chosen output Format **/
18     protected VideoFormat outputFormat;
19
20     /** supported input Formats **/
21     protected Format[] supportedInputFormats=new Format[0];
22
23     /** supported output Formats **/
24     protected Format[] supportedOutputFormats=new Format[0];
25
26     // Buffer number
27     int buffNbr;
28
29     // Hash size
30     int hashSize = 16;
31
32     // Number of buffer per block
33     public int blockSize = 8;
34
35     // The hasher
36     MessageDigest digester = null;
37
38     // JCA variables
39     PublicKey pubKey;
40     Signature dsa;
41
42     /** initialize the plugin **/
43     public H263VideoVerifierStar() {
44
45         // Set the input and output formats of this plugin
46         supportedInputFormats = new Format[] {
47             new VideoFormat(VideoFormat.H263_RTP)
48         };
49         supportedOutputFormats = new Format[] {
50             new VideoFormat(VideoFormat.H263_RTP)
51         };
52
53         // Initialize the hasher
54         try {
55             digester = MessageDigest.getInstance("MD5");
56         }
57         catch(NoSuchAlgorithmException e) {
58             error("No Such Algorithm Exception");
59         }
60
61         // Read the public key
62         File f = new File("./pubKey.data");
63
64         FileInputStream fis = null;
65
66         try {
67             fis = new FileInputStream(f);
68         }
69         catch(FileNotFoundException e) {
70             error("File Not Found Exception");
71         }
72
73         try {
74             ObjectInput ois = new ObjectInputStream(fis);
75
76             try {
77                 pubKey = (PublicKey) ois.readObject();
78             }
79             catch(ClassNotFoundException e) {
80                 error("Class Not Found Exception");
```

```
81              }
82
83              ois.close();
84
85              info("Public Key File read");
86          }
87          catch(IOException e) {
88              error("I/O Exception");
89          }
90
91          // Signature verification initialization
92          try {
93              dsa = Signature.getInstance("SHA1withDSA");
94              dsa.initVerify(pubKey);
95          }
96          catch(NoSuchAlgorithmException e) {
97              error("No Such Algorithm Exception");
98          }
99          catch(InvalidKeyException e) {
100             error("Invalid Key Exception");
101         }
102     }
103
104     /** get the resources needed by this plugin **/
105     public void open() throws ResourceUnavailableException {
106     }
107
108     /** free the resources allocated by this plugin **/
109     public void close() {
110     }
111
112     /** reset the plugin **/
113     public void reset() {
114     }
115
116     /** no controls for this simple plugin **/
117     public Object[] getControls() {
118         return (Object[]) new Control[0];
119     }
120
121     /** Return the control based on a control type for the plugin **/
122     public Object getControl(String controlType) {
123         try {
124             Class cls = Class.forName(controlType);
125             Object cs[] = getControls();
126             for (int i = 0; i < cs.length; i++) {
127                 if (cls.isInstance(cs[i]))
128                     return cs[i];
129             }
130             return null;
131         } catch (Exception e) { // no such controlType or such control
132             return null;
133         }
134     }
135
136     /************** format methods *************/
137     /** set the input format **/
138     public Format setInputFormat(Format input) {
139         // the following code assumes valid Format
140         inputFormat = (VideoFormat)input;
141         return (Format)inputFormat;
142     }
143
144     /** set the output format **/
145     public Format setOutputFormat(Format output) {
146         // the following code assumes valid Format
147         outputFormat = (VideoFormat)output;
148         return (Format)outputFormat;
149     }
150
151     /** get the input format **/
152     protected Format getInputFormat() {
153         return inputFormat;
154     }
155
156     /** get the output format **/
157     protected Format getOutputFormat() {
158         return outputFormat;
159     }
160
161     /** supported input formats **/
162     public Format [] getSupportedInputFormats() {
163         return supportedInputFormats;
```

```
164     }
165
166     /** output Formats for the selected input format **/
167     public Format [] getSupportedOutputFormats(Format in) {
168         if (in == null)
169             return supportedOutputFormats;
170         else {
171             Format outs[] = new Format[1];
172             outs[0] = in;
173             return outs;
174         }
175     }
176
177     /** return plugin name **/
178     public String getName() {
179         return CodecName;
180     }
181
182     /** do the processing **/
183     public int process(Buffer inputBuffer, Buffer outputBuffer){
184
185         // Get the input buffer info
186         int inLength = inputBuffer.getLength();
187         int inOffset = inputBuffer.getOffset();
188         byte[] inData =
189             getValidData((byte[]) inputBuffer.getData(),inOffset,inLength);
190
191         // Retrieve the block signature from the buffer
192         int sigSize = getSigSize(inData);
193         byte[] sig = extractSignature(inData,sigSize);
194         // Retrieve the block number to which this buffer belongs
195         int blockNbr = extractBlockNbr(inData,sigSize);
196         // Retrieve the current block position in the current block
197         int posInBlock = extractPosition(inData,sigSize);
198         // Retrieve the buffer number
199         buffNbr = extractBuffNbr(inData,sigSize);
200
201         // Compute the length of the video data contained in this buffer
202         int videoDataLength =
203             inData.length - 1 - sigSize - 4 - 4 - ((blockSize - 1) * hashSize) - 4;
204
205         // Print some info
206         info("*****************************");
207         info("Buffer #" + buffNbr);
208         info("Input buffer length: " + inLength);
209         info("Format/encoding : " + inputBuffer.getFormat().getEncoding());
210         info("Signature size: " + sigSize);
211         info("Block number: " + blockNbr);
212         info("Position in block: " + posInBlock);
213
214         // Compute current buffer hash
215         int i;
216         byte[] dataTmp = new byte[videoDataLength + 4];
217         for(i=0;i<videoDataLength+4;i++)
218             dataTmp[i] = inData[i];
219         byte[] buffHash = computeHash(dataTmp);
220
221         // Extract signed portion
222         dataTmp = extractSignedData(inData,videoDataLength+4,posInBlock,buffHash);
223
224         // Verify signature
225         if(verifSignature(computeHash(dataTmp),sig))
226             info("Signature verification ok");
227         else
228             error("Bad signature");
229
230         // Extract the video data from this buffer
231         byte[] outData = extractVideoData(inData,videoDataLength);
232
233         // Set the content and properties of the output buffer
234         outputBuffer.setData(outData);
235         outputBuffer.setFormat(inputBuffer.getFormat());
236         outputBuffer.setLength(outData.length);
237         outputBuffer.setOffset(0);
238
239         return BUFFER_PROCESSED_OK;
240     }
241
242     // Extract the valid video data from the input buffer
243     private byte[] getValidData(byte[] data,int offset,int length) {
244
245         byte[] tmp = new byte[length];
246         int i;
```

```
247
248         for(i=0;i<length;i++)
249             tmp[i] = data[offset + i];
250
251         return tmp;
252     }
253
254     // Extract the buffer number
255     private int extractBuffNbr(byte[] data, int sigSize) {
256
257         byte[] tmp = new byte[4];
258         int i;
259         int offset =
260             data.length - 1 - sigSize - 4 - 4 - ((blockSize - 1) * hashSize) - 4;
261
262         for(i=0;i<4;i++)
263             tmp[i] = data[offset + i];
264
265         return byteArray2Int(tmp);
266     }
267
268     // Compute the signature size
269     private int getSigSize(byte[] data) {
270         return (int) (data[data.length - 1]);
271     }
272
273     // Extract the signature from a buffer
274     private byte[] extractSignature(byte[] data,int sigSize) {
275
276         byte[] sig = new byte[sigSize];
277         int offset = data.length - 1 - sigSize;
278         int i;
279         for(i=0;i<sigSize;i++)
280             sig[i] = data[offset + i];
281         return sig;
282     }
283
284     // Extract the block number of a buffer
285     private int extractBlockNbr(byte[] data,int sigSize) {
286
287         byte[] blockNbr = new byte[4];
288         int offset = data.length - 1 - sigSize - 4;
289         int i;
290         for(i=0;i<4;i++)
291             blockNbr[i] = data[offset + i];
292         return byteArray2Int(blockNbr);
293     }
294
295     // Extract the position of a buffer in a block
296     private int extractPosition(byte[] data,int sigSize) {
297
298         byte[] position = new byte[4];
299         int offset = data.length - 1 - sigSize - 4 - 4;
300         int i;
301         for(i=0;i<4;i++)
302             position[i] = data[offset + i];
303         return byteArray2Int(position);
304     }
305
306     // Convert a byte array to an integer
307     private int byteArray2Int(byte[] val) {
308         int tmp = 0;
309         int i;
310         for (i=3;i>0;i--) {
311             tmp = tmp + ((int) val[i] & (int) 0xff);
312             tmp = tmp << 8;
313         }
314         tmp = tmp + ((int) val[0] & (int) 0xff);
315         return tmp;
316     }
317
318     // Extract the video data from a buffer
319     private byte[] extractVideoData(byte[] data, int length) {
320
321         byte[] tmp = new byte[length];
322         int i;
323
324         for(i=0;i<length;i++)
325             tmp[i] = data[i];
326
327         return tmp;
328     }
329
```

```
330     // Compute current buffer hash
331     private byte[] computeHash(byte[] data) {
332         return digester.digest(data);
333     }
334 
335     // Extract the signed portion of the buffer
336     private byte[] extractSignedData(byte[] data, int offset,
337                                      int posInBlock, byte[] buffHash) {
338 
339         byte[] result = new byte[blockSize * hashSize];
340         int i,j;
341 
342         // Copy the first "posInBlock" hash(es)
343         for(i=0;i<(posInBlock*hashSize);i++)
344             result[i] = data[offset + i];
345 
346         // Copy current hash
347         for(j=0;j<hashSize;j++)
348             result[i + j] = buffHash[j];
349 
350         // Copy the last hashes
351         while((i + hashSize) < result.length) {
352             result[i + hashSize] = data[offset + i];
353             i++;
354         }
355 
356         return result;
357     }
358 
359     // Verify the signature of a buffer
360     private boolean verifSignature(byte[] data, byte[] sig) {
361 
362         boolean result = false;
363 
364         try {
365             dsa.update(data);
366             result = dsa.verify(sig);
367         }
368         catch(SignatureException e) {
369             error("Signature Exception");
370         }
371         return result;
372     }
373 
374     private void info(String msg) {
375         System.out.println("[INFO] H263 Video Verifier Plugin: " + msg);
376     }
377 
378     private void error(String msg) {
379         System.err.println("[ERROR] H263 Video Verifier Plugin: " + msg);
380         System.exit(1);
381     }
382 }
```

Simulator/Server/Plugin/H263VideoVerifierTree.java

```
1  package Server.Plugin;
2
3  import java.io.*;
4  import java.util.*;
5  import javax.media.*;
6  import javax.media.format.*;
7  import javax.media.format.AudioFormat;
8  import java.security.*;
9
10 public class H263VideoVerifierTree implements Codec {
11
12     private static String CodecName="H263VideoVerifierTree";
13
14     /** chosen input Format **/
15     protected VideoFormat inputFormat;
16
17     /** chosen output Format **/
18     protected VideoFormat outputFormat;
19
20     /** supported input Formats **/
21     protected Format[] supportedInputFormats=new Format[0];
22
23     /** supported output Formats **/
24     protected Format[] supportedOutputFormats=new Format[0];
25
26     // Buffer number
27     int buffNbr;
28
29     // Hash size
30     int hashSize = 16;
31
32     // Number of buffer per block
33     public int blockSize = 8;
34
35     // Tree degree such that log(treeDeg)(blockSize) == belongs to the int set
36     public int treeDeg = 2;
37
38     // Tree height
39     private int treeHeight;
40
41     // Number of hashes in a buffer
42     int hashQuantity;
43
44     // The hasher
45     MessageDigest digester = null;
46
47     // JCA variables
48     PublicKey pubKey;
49     Signature dsa;
50
51     /** initialize the plugin **/
52     public H263VideoVerifierTree() {
53
54         // Compute some parameters
55         hashQuantity =
56             (treeDeg - 1) *
57             (int) (java.lang.Math.log(blockSize) / java.lang.Math.log(treeDeg));
58         info("Number of hashes contained in input buffers: " + hashQuantity);
59
60         treeHeight =
61             (int) (java.lang.Math.log(blockSize) / java.lang.Math.log(treeDeg));
62         info("Tree height: " + treeHeight);
63
64         // Set the input and output formats of this plugin
65         supportedInputFormats = new Format[] {
66             new VideoFormat(VideoFormat.H263_RTP)
67         };
68         supportedOutputFormats = new Format[] {
69             new VideoFormat(VideoFormat.H263_RTP)
70         };
71
72         // Initialize the hasher
73         try {
74             digester = MessageDigest.getInstance("MD5");
75         }
76         catch(NoSuchAlgorithmException e) {
77             error("No Such Algorithm Exception");
78         }
79
80         // Read the public key
```

```
81          File f = new File("./pubKey.data");
82
83          FileInputStream fis = null;
84
85          try {
86              fis = new FileInputStream(f);
87          }
88          catch(FileNotFoundException e) {
89              error("File Not Found Exception");
90          }
91
92          try {
93              ObjectInput ois = new ObjectInputStream(fis);
94
95              try {
96                  pubKey = (PublicKey) ois.readObject();
97              }
98              catch(ClassNotFoundException e) {
99                  error("Class Not Found Exception");
100             }
101
102             ois.close();
103
104             info("Public Key File read");
105         }
106         catch(IOException e) {
107             error("I/O Exception");
108         }
109
110         // Signature verification initialization
111         try {
112             dsa = Signature.getInstance("SHA1withDSA");
113             dsa.initVerify(pubKey);
114         }
115         catch(NoSuchAlgorithmException e) {
116             error("No Such Algorithm Exception");
117         }
118         catch(InvalidKeyException e) {
119             error("Invalid Key Exception");
120         }
121     }
122
123     /** get the resources needed by this plugin **/
124     public void open() throws ResourceUnavailableException {
125     }
126
127     /** free the resources allocated by this plugin **/
128     public void close() {
129     }
130
131     /** reset the plugin **/
132     public void reset() {
133     }
134
135     /** no controls for this simple plugin **/
136     public Object[] getControls() {
137         return (Object[]) new Control[0];
138     }
139
140     /** Return the control based on a control type for the plugin **/
141     public Object getControl(String controlType) {
142         try {
143             Class cls = Class.forName(controlType);
144             Object cs[] = getControls();
145             for (int i = 0; i < cs.length; i++) {
146                 if (cls.isInstance(cs[i]))
147                     return cs[i];
148             }
149             return null;
150         } catch (Exception e) { // no such controlType or such control
151             return null;
152         }
153     }
154
155     /************** format methods *************/
156     /** set the input format **/
157     public Format setInputFormat(Format input) {
158         // the following code assumes valid Format
159         inputFormat = (VideoFormat)input;
160         return (Format)inputFormat;
161     }
162
163     /** set the output format **/
```

```
164     public Format setOutputFormat(Format output) {
165         // the following code assumes valid Format
166         outputFormat = (VideoFormat)output;
167         return (Format)outputFormat;
168     }
169 
170     /** get the input format **/
171     protected Format getInputFormat() {
172         return inputFormat;
173     }
174 
175     /** get the output format **/
176     protected Format getOutputFormat() {
177         return outputFormat;
178     }
179 
180     /** supported input formats **/
181     public Format [] getSupportedInputFormats() {
182         return supportedInputFormats;
183     }
184 
185     /** output Formats for the selected input format **/
186     public Format [] getSupportedOutputFormats(Format in) {
187         if (in == null)
188             return supportedOutputFormats;
189         else {
190             Format outs[] = new Format[1];
191             outs[0] = in;
192             return outs;
193         }
194     }
195 
196     /** return plugin name **/
197     public String getName() {
198         return CodecName;
199     }
200 
201     /** do the processing **/
202     public int process(Buffer inputBuffer, Buffer outputBuffer){
203 
204         // Get the input buffer info
205         int inLength = inputBuffer.getLength();
206         int inOffset = inputBuffer.getOffset();
207         byte[] inData =
208             getValidData((byte[]) inputBuffer.getData(),inOffset,inLength);
209 
210         // Retrieve the block signature from the buffer
211         int sigSize = getSigSize(inData);
212         byte[] sig = extractSignature(inData,sigSize);
213         // Retrieve the block number to which this buffer belongs
214         int blockNbr = extractBlockNbr(inData,sigSize);
215         // Retrieve the current block position in the current block
216         int posInBlock = extractPosition(inData,sigSize);
217         // Retrieve the buffer number
218         buffNbr = extractBuffNbr(inData,sigSize);
219 
220         // Compute the length of the video data contained in this buffer
221         int videoDataLength =
222             inData.length - 1 - sigSize - 4 - 4 - (hashQuantity * hashSize) - 4;
223 
224         // Print some info
225         info("*******************************");
226         info("Buffer #" + buffNbr);
227         info("Input buffer length: " + inLength);
228         info("Format/encoding : " + inputBuffer.getFormat().getEncoding());
229         info("Signature size: " + sigSize);
230         info("Block number: " + blockNbr);
231         info("Position in block: " + posInBlock);
232 
233         // Compute current buffer hash
234         int i;
235         byte[] dataTmp = new byte[videoDataLength + 4];
236         for(i=0;i<videoDataLength+4;i++)
237             dataTmp[i] = inData[i];
238         byte[] buffHash = computeHash(dataTmp);
239 
240         // Extract signed portion
241         dataTmp = extractSignedData(inData,videoDataLength+4,posInBlock,buffHash);
242 
243         // Verify signature
244         if(verifSignature(dataTmp,sig))
245             info("Signature verification ok");
246         else
```

```
247             error("Bad signature");
248
249         // Extract the video data from this buffer
250         byte[] outData = extractVideoData(inData,videoDataLength);
251
252         // Set the content and properties of the output buffer
253         outputBuffer.setData(outData);
254         outputBuffer.setFormat(inputBuffer.getFormat());
255         outputBuffer.setLength(outData.length);
256         outputBuffer.setOffset(0);
257
258         return BUFFER_PROCESSED_OK;
259     }
260
261     // Extract the valid video data from the input buffer
262     private byte[] getValidData(byte[] data,int offset,int length) {
263
264         byte[] tmp = new byte[length];
265         int i;
266
267         for(i=0;i<length;i++)
268             tmp[i] = data[offset + i];
269
270         return tmp;
271     }
272
273     // Extract the buffer number
274     private int extractBuffNbr(byte[] data, int sigSize) {
275
276         byte[] tmp = new byte[4];
277         int i;
278         int offset =
279             data.length - 1 - sigSize - 4 - 4 - (hashQuantity * hashSize) - 4;
280
281         for(i=0;i<4;i++)
282             tmp[i] = data[offset + i];
283
284         return byteArray2Int(tmp);
285     }
286
287     // Compute the signature size
288     private int getSigSize(byte[] data) {
289         return (int) (data[data.length - 1]);
290     }
291
292     // Extract the signature from a buffer
293     private byte[] extractSignature(byte[] data,int sigSize) {
294
295         byte[] sig = new byte[sigSize];
296         int offset = data.length - 1 - sigSize;
297         int i;
298         for(i=0;i<sigSize;i++)
299             sig[i] = data[offset + i];
300         return sig;
301     }
302
303     // Extract the block number of a buffer
304     private int extractBlockNbr(byte[] data,int sigSize) {
305
306         byte[] blockNbr = new byte[4];
307         int offset = data.length - 1 - sigSize - 4;
308         int i;
309         for(i=0;i<4;i++)
310             blockNbr[i] = data[offset + i];
311         return byteArray2Int(blockNbr);
312     }
313
314     // Extract the position of a buffer in a block
315     private int extractPosition(byte[] data,int sigSize) {
316
317         byte[] position = new byte[4];
318         int offset = data.length - 1 - sigSize - 4 - 4;
319         int i;
320         for(i=0;i<4;i++)
321             position[i] = data[offset + i];
322         return byteArray2Int(position);
323     }
324
325     // Convert a byte array to an integer
326     private int byteArray2Int(byte[] val) {
327         int tmp = 0;
328         int i;
329         for (i=3;i>0;i--) {
```

```
330             tmp = tmp + ((int) val[i] & (int) 0xff);
331             tmp = tmp << 8;
332         }
333         tmp = tmp + ((int) val[0] & (int) 0xff);
334         return tmp;
335     }
336
337     // Extract the video data from a buffer
338     private byte[] extractVideoData(byte[] data, int length) {
339
340         byte[] tmp = new byte[length];
341         int i;
342         for(i=0;i<length;i++)
343             tmp[i] = data[i];
344
345         return tmp;
346     }
347
348     // Compute current buffer hash
349     private byte[] computeHash(byte[] data) {
350         return digester.digest(data);
351     }
352
353     // Extract the signed portion of the buffer
354     private byte[] extractSignedData(byte[] data, int offset,
355                                      int posInBlock, byte[] buffHash) {
356
357         int i,j,k,dataTmpOffset;
358         byte[] dataTmp;
359         byte[] currentHash = buffHash;
360         int posInLevel = posInBlock;
361         int posInBranch = posInBlock % treeDeg;
362
363         for(i=0;i<treeHeight;i++) {
364             dataTmp = new byte[treeDeg * hashSize];
365
366             for(j=0;j<treeDeg;j++) {
367                 dataTmpOffset = j * hashSize;
368                 if(j == posInBranch) {
369                     // Copy currentHash into dataTmp
370                     for(k=0;k<hashSize;k++)
371                         dataTmp[dataTmpOffset + k] = currentHash[k];
372                 }
373                 else {
374                     // Copy hash from data into dataTmp
375                     for(k=0;k<hashSize;k++)
376                         dataTmp[dataTmpOffset + k] = data[offset + k];
377                     offset+=hashSize;
378                 }
379             }
380
381             currentHash = computeHash(dataTmp);
382             posInLevel = posInLevel / treeDeg;
383             posInBranch = posInLevel % treeDeg;
384         }
385         return currentHash;
386     }
387
388     // Verify the signature of a buffer
389     private boolean verifSignature(byte[] data, byte[] sig) {
390
391         boolean result = false;
392         try {
393             dsa.update(data);
394             result = dsa.verify(sig);
395         }
396         catch(SignatureException e) {
397             error("Signature Exception");
398         }
399         return result;
400     }
401
402     private void info(String msg) {
403         System.out.println("[INFO] H263 Video Verifier Plugin: " + msg);
404     }
405
406     private void error(String msg) {
407         System.err.println("[ERROR] H263 Video Verifier Plugin: " + msg);
408         System.exit(1);
409     }
410 }
```

Bibliography

[1] Anonymous. *FIPS 180-1, Secure Hash Standard.* National Institute of Standards and Technology, US Department of Commerce, Washington, DC, USA, Apr. 1995.

[2] Anonymous. *Java Media Framework API guide.* Sun Microsystems, Nov. 1999.

[3] O. Bonaventure. Réseaux: Matières approfondies. Lecture Notes, Facultés Universitaires Notre-Dame de la Paix, Namur, 2001.

[4] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. *Sigcom 90*, pages 200–208, 1990.

[5] H. M. Deitel and P. J. Deitel. *Java: how to program.* How to program series. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, fourth edition, 2002. CD-ROM contains Java TM 2 SDK, Standard Edition, 1.3.1, Java Media Framework API 2.1.1, Forte for Java, Release 2.0, Community Edition and Java Plug-in HTML Converter 1.3.

[6] D. Eastlake and P. Jones. RFC 3174: US secure hash algorithm 1 SHA1, Sept. 2001. Status: INFORMATIONAL.

[7] R. Gennaro and P. Rohatgi. How to sign digital streams. *Lecture Notes in Computer Science*, 1294, 1997.

[8] P. Golle and N. Modadugu. Authenticating streamed data in the presence of random packet loss. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2001)*, pages 13–22, San Diego, CA, Feb. 2001. Internet Society.

[9] A.-V. T. W. Group and H. Schulzrinne. RFC 1890: RTP profile for audio and video conferences with minimal control, Jan. 1996. Status: PROPOSED STANDARD.

[10] A.-V. T. W. Group, H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC 1889: RTP: A transport protocol for real-time applications, Jan. 1996. Status: PROPOSED STANDARD.

[11] B. Kaliski. RFC 2313: PKCS #1: RSA encryption version 1, Mar. 1998. Status: INFORMATIONAL.

[12] B. Kaliski and J. Staddon. RFC 2437: PKCS #1: RSA cryptography specifications version 2.0, Oct. 1998. Obsoletes RFC2313 [11]. Status: INFORMATIONAL.

[13] A. J. A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography.* The CRC Press series on discrete mathematics and its applications. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1997.

[14] S. Miner and J. Staddon. Graph-based authentication of digital streams. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 232–246, Oakland, CA, May 2001. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[15] National Institute of Standards and Technology (NIST). The Digital Signature Standard (DSS). FIPS PUB 186-2, Jan. 2000.

[16] A. Perrig, R. Canetti, D. Tygar, and D. Song. Efficient authentication and signature of multicast streams over lossy channels. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 2000. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[17] T. Peuker. An object-oriented architecture for the real-time transmission of multimedia data streams. Lehrstuhl für betriebssyteme universität erlangen-nürnberg, Institut für Mathematische Maschinen und Datenverarbeitung (Informatik) IV, 1997.

[18] R. Rivest. RFC 1321: The MD5 message-digest algorithm, Apr. 1992. Status: INFORMATIONAL.

[19] Rohatgi. A compact and fast hybrid signature scheme for multicast packet authentication. In *SIGSAC: 6th ACM Conference on Computer and Communications Security.* ACM SIGSAC, 1999.

[20] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley and Sons, Inc., New York, NY, USA, second edition, 1996.

[21] C. K. Wong and S. S. Lam. Digital signatures for flows and multicasts. In *IEEE ICNP '98*, 1998.